Covert Surveillance on PC and Android

Szymon Sidor

Predrag Gruevski

David Xiao

Arash Delijani

6.857, May 2014

1 - Introduction

Modern computers are far more capable, ubiquitous, and integrated into our lives than a generation ago. The cheapest iPhone today contains more processing power than the fastest supercomputer in the world twenty years ago, as well as a wifi card, 4G receiver, camera, microphone, and speakers to boot. This has made activities such as sending friends pictures of your latest kayaking misadventures or paying off your phone bill easier and faster than ever. However, having all that capability and personal information in your pocket or briefcase can also be a serious privacy and security risk. New ways of stealing your credit card information or destroying uranium centrifuges are published almost daily. The aim of this project was to investigate existing methods and discover new methods for covertly surveilling user activities using their compromised PCs and smartphones. While we did not find any new technical vulnerabilities, we did find human interface and design vulnerabilities in PC and Android smartphones.

The information we are interested in gathering from a PC and smartphone is focused not on activities on the device, but on activities in the real world. Methods for logging all of a user's keystrokes, scanning their hard drive for personal information, capturing images of their screen, or hijacking their email are all related to their interactions with the computer. The existence of cameras, microphones, and location services on most devices, however, allow an attacker to surveil their interactions in the real world. This could include their current location on the 2nd floor lounge of a corporate office, the argument they are having with their boss, maybe even the exact pattern of their boss's tie.

Most devices, especially smartphones, have built-in protections to prevent applications from accessing sensors (such as the camera) which users did not approve them to use, and alert the user in some way when a sensor is activated. Android applications must list at install time any sensors they intend to use. iPhones display a little arrow in the corner of the screen when determining the current location. All laptops and tablets have an indicator light next to the built-in camera, which turns on when the camera is active.

However, in some cases it may be possible to circumvent these safeguards. One such well-publicized event occurred in 2013, when a 19-year-old pleaded guilty to extortion for hacking into the webcams of 12 women, taking compromising photos of them, and then blackmailing them with the photos. None of the women, one of whom was a former Miss Teen USA, reported ever seeing the indicator light of their webcam turn on.

Two methods of attacking webcams formed the foundation and initial motivation for this project. The first, Metasploit, is a well known penetration-testing software suite which contains a method for exploiting a computer's camera to take pictures computer's location.[1] The command, *webcam_snap*, is located in the meterpreter module, and allows an attacker to access any

[1] "Meterpreter Basics" *Offensive Security*. Accessed: 5/11/2014 <http://www.offensive-security.com/metasploit-unleashed/Meterpreter_Basics>

camera attached to the compromised computer, briefly turn the camera on, and get an image of the computer's surroundings and possibly of the user. This attack does not deactivate the camera indicator light, but instead activates the camera and light for a very brief amount of time (fractions of a second) in the hope that the user will not notice. Testing this assumption formed the basis for one of the three parts of this project.

The second method is iSeeYou, an attack published in late 2013 which exploits a hardware vulnerability in 2008-era MacBooks to disable the camera indicator light entirely.[2] A design flaw in the hardware interlock between the camera and its indicator light allowed a signal from the firmware to suppress the indicator light even when the camera was active. Since firmware could be reprogrammed via driver update initiated by a user-space process, the attack could be done without even needing root access on the machine. Searching for other ways of recording video on PCs and smartphones without giving any indication to users formed the other two thirds of this project.

Our project consists of three parts, described in sections 2-4:

- **Part 1** details our attempts to discover backdoors into the USB cameras embedded in our laptops via usb packet sniffing and injection. While the attempts provided valuable experience in working with usb devices at a low level, we were unable to find attack vectors and determined that further attempts would be both too computationally intensive and unlikely to yield useful results.

- **Part 2** was human-centric -- we conducted a user study to determine how noticeable the camera indicator light was for users under real-world cognitive load. Results varied depending on the device's indicator light and users' engagement with their current tasks, but showed that under the right conditions, humans can be made to reliably fail to notice the indicator light activity.

- **Part 3** shifted its focus to the smartphone world, investigating the possibility of applications capturing data from an Android device without the user noticing. We discovered that it is possible for an application to record video from a device with only a single 1x1 pixel overlay on the screen serving as an "indicator" of the application's activity to the user. Since a single pixel is practically invisible on modern phones with pixel densities of 400+dpi, this allows an application to capture pictures or video *without any indication to the user*. This is, in our opinion, a serious security risk on the Android platform, and should be remedied as soon as possible.

In section 5, we discuss possible future research directions stemming from this project.

[2] Matthew Brocker and Stephen Checkoway. 2013. iSeeYou: Disabling the MacBook Webcam Indicator LED. *Current Technical Reports, Johns Hopkins University Dept. of Computer Science.* (December 2013) DOI: http://jhir.library.jhu.edu/handle/1774.2/36569

2 - Exploiting USB protocols in laptop webcams

The goal of this part of the project was to reverse engineer the usb protocol used by the webcam driver and examine whether we can modify the protocol in a way that would cause the webcam to misbehave.

The specific setup we used for that part of the project was:

Computer	Lenovo Thinkpad W530
Webcam	built-in Chicony Electronics Co., Ltd Integrated Camera
Operating System	GNU/Linux Fedora 20

2.1 - Our approach

There are many ways one can reverse-engineer a USB device driver:

- inspecting the source code directly;
- decompiling the driver;
- inspecting raw data exchanged between the driver and the device.

We chose the last of these approaches. This decision was mostly made because of the fact that the driver source code is available as part of the huge video for linux package, which is in turn part of the Linux kernel. The raw interaction with the USB camera is quite simple compared to complexity of that package.

2.2 - USB in GNU/Linux operating system

We began by examining what steps are involved in the interaction between Linux applications and a webcam. Our findings are summarized in the diagram below:

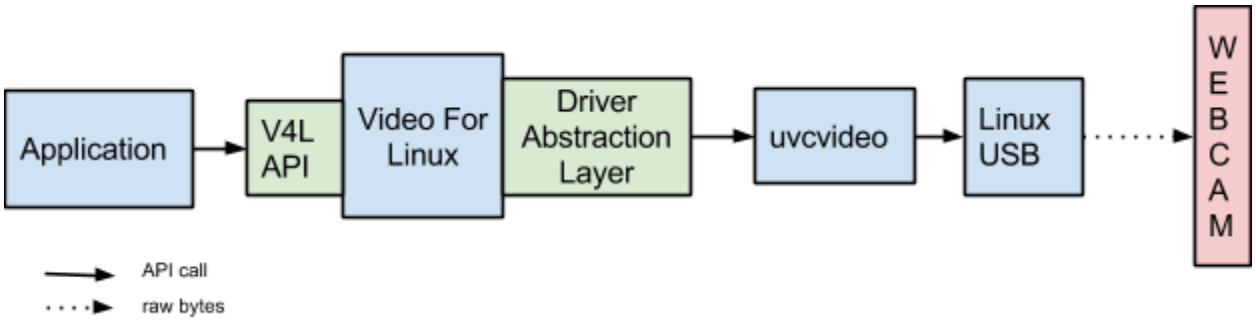


Figure 1: Software chain for controlling a webcam.

Applications use the Video For Linux (v4l) API to communicate with the camera. The purpose of v4l is to abstract the complexities of dealing with multiple different camera makes and models. V4l makes API calls to the driver abstraction layer (inside v4l) and, in case of our camera, the uvcvideo driver is invoked. This driver interfaces with the Linux USB drivers and sends raw bytes to the webcam.

The driver's code is running in privileged mode, because it needs to interact directly with the hardware.

2.3 - Overview of USB protocols

The first step towards reverse-engineering a USB-based protocol was to familiarize ourselves with the details of the protocol as it exists between the USB device and the host operating system.

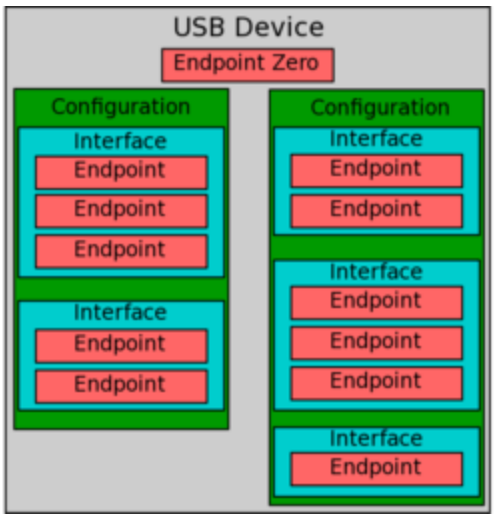


Figure 2: USB device endpoint organization.

To communicate with a USB device, one needs to know its vendor ID and product ID. After identifying the device, there are certain setup steps that have to be invoked: each USB device has a certain number of endpoints -- we can think about them as elements of its API. One of the endpoints is special -- endpoint zero -- it is used for control traffic. The USB protocol was designed with flexibility and efficiency in mind, so instead of just having a list of endpoints, we have multiple levels of abstraction:

- **Configuration:** before we interact with device we need to choose in what configuration to put it. For example, for a webcam, configuration 0 could be a photo mode and configuration 1 could be video mode. Each configuration has multiple interfaces.
- **Interface:** another level of abstraction used to group some endpoints together. For example, if we have a camera on a robotic arm, we could use interface 0 to get the data from the camera and interface 1 to control the arm. Each interface can have multiple endpoints, and represents a similar abstraction level as classes in source code.

- **Endpoint:** this is a logical address used to send or receive data to and from device. For example, for a robotic arm, we can have one endpoint per joint that we control. One can think about endpoints as functions in source code.
- **Alternative settings:** each interface can have multiple alternative settings which change the internal mode of operation of that interface. For example, for a robotic arm, the control interface could have two sets of alternative settings -- one where the arm is controlled by specifying angles between joints, and another where the user provides cartesian coordinates relative to the robot. Alternative settings are a bit like parameters passed to the constructor of a class in source code.

Once we determine which configuration/interfaces/alternative settings/endpoints we are going to use, we use endpoint zero to inform the device, and then we are ready to talk to an endpoint. There are four ways one can talk to an endpoint, although each particular endpoint can only talk in one way:

- **Control:** used for sending commands to the device -- those are usually short packets with immediate status reply from the device. Example: setting webcam brightness.
- **Interrupt:** continuous polling of data from the device with a guaranteed latency bound. Example: motion detector informing the OS that it detected motion.
- **Isochronous:** high speed streaming of large quantities of data. No automated retry or guaranteed delivery, so a bit like UDP. Example: video or audio traffic.
- **Bulk:** high speed streaming of large quantities of data. Automated reply and guaranteed delivery, so a bit like TCP. Example: file transfers.

Now that we established the basics, we can move to a practical example of reverse engineering a USB-based protocol.

2.4 - Reverse Engineering a particular USB protocol

First and foremost, we need to learn the vendor ID and product ID of our device. This can be done by issuing the **lsusb** command, which will list out multiple lines, one line per USB device. The line related to our camera looked like this:

```
Bus 001 Device 004: ID 04f2:b2ea Chicony Electronics Co., Ltd
Integrated Camera [ThinkPad]
```

Here, 04f2 is the vendor ID and b2ea is the product ID. If you have trouble identifying which line corresponds to your device, you can try running **lsusb** with and without the device plugged in. It's also worth noting the bus (001) and the device (004). Having established that basic information, we can use the **usbmon** utility to monitor device traffic, understand and reverse engineer it. In the case of Fedora 20, **usbmon** comes pre-installed. In true GNU/Linux fashion, one can look at the traffic by reading a from a file as follows:

```
cat /sys/kernel/debug/usb/usbmon/1u | grep ":004:"
```

Here, 1 and 004 correspond to the bus and device numbers. This command will only produce output if the camera is in use. Here's a subset of the output we got:

```
1. ffff88011afa79c0 3449979436 S Co:004:00 s 01 0b 0005 0001 0000 0
2. ffff880221f36c00 3449979738 S Zi:004:02 -115 98304 <
3. ffff880221f33c00 3449979745 S Zi:004:02 -115 98304 <
4. ffff880221f35800 3449979748 S Zi:004:02 -115 98304 <
5. ffff880221f33800 3449979751 S Zi:004:02 -115 98304 <
6. ffff880221f30400 3449979753 S Zi:004:02 -115 98304 <
7. ffff880221f36c00 3449984651 C Zi:004:02 0 98304 = 00000000 00000000 00000000
00000000 00000000 00000000 00000000 00000000
8. ffff880221f36c00 3449984656 S Zi:004:02 -115 98304 <
9. ffff880221f33c00 3449988659 C Zi:004:02 0 98304 = 00000000 00000000 00000000
00000000 00000000 00000000 00000000 00000000
...
X. ffff88011b5c0480 3844817942 S Co:1:004:0 s 01 0b 0000 0001 0000 0
```

While it might look a bit overwhelming at first, it's not that difficult to interpret. Let's take the first line and analyze its structure:

ffff88011afa79c0	**3449979436**	**S**	**Co**	**004**	**00**
URB tag - use for identifying requests	timestamp in microsections	request type	type of transfer	device number	endpoint number

01	**0b**	**0005**	**0001**	**0**
bmRequestType - direction (to or from device), and type of request and logical recipient	bRequest - request type[3]	wValue - command value	wIndex - command indexi	wLength - length of data passed

Figure 3: Breakdown of a USB protocol packet.

Armed with knowledge how to interpret the data, we can proceed to understanding the protocol. Line 1 is a control command instructing camera that we are about to request camera feed. Lines 2-6 are isochronous requests for data, which are initially empty because the data is not ready. The camera LED turns on when executing those lines. Lines 7 to X-1 are further isochronous transfers that actually output data, and finally, line X issues control data to stop the transfer, and the camera LED turns off.

[3] "9.2 USB Control Transfers - Jungo." Accessed: 13 May. 2014 <http://www.jungo.com/st/support/documentation/windriver/10.2.1/wdusb_manual.mhtml/USB_Control_Trans.html>

2.5 - Using pyusb to talk to USB devices

Now that we learned some details about the protocol, let's try and see if we can redo it by ourselves. The easiest way to understand how this can be done is by looking at a code sample:

```
# Those are the values we found with lsusb
device = usb.core.find(idVendor=0x04f2, idProduct=0xb2ea)
# In Linux, when the kernel loads the driver,
# usb interaction with the device is blocked,
# so we need to detach it.
try:
    device.detach_kernel_driver(0)
except Exception as e:
    print 'already unregistered'
# Set device to default configuration.
device.set_configuration()
```

First verified that we can achieve something simple. The following line was intercepted by **usbmon** when changing the webcam brightness to 47:

```
ffff8800895299c0 2236018417 S Co:004:00 s 21 01 0200 0200 0002 2
= 2f00
```

Here `2f` corresponds to 47. We successfully replayed this command by executing this Python:

```
device.ctrl_transfer(0x21, 0x01, 0x0200, 0x0200, [47,00])
```

We can then verify that the brightness of the image changed indeed. Getting the image off the camera was a bit more involved:

```
# Set device to default configuration (no. 0)
device.set_configuration()
configuration = device.get_active_configuration()
# set interface 1 to alternative settings no. 1
device.set_interface_altsetting(1,1)
interface = configuration[(0,0)]
# get the 0-th endpoint of that interface
endpoint = interface[0]
# Tell the device to prepare for video capture (*)
device.ctrl_transfer(0x01, 0x0b, 0x0005, 0x0001, [])
# Initiate isochronous transfer from camera limiting returned
# data to 98304 bits. This will not return any data because it
# takes about a second for any data to be available. LED will
# turn on at this point.
endpoint.read(98304)
# sleep a second.
time.sleep(1)
# This will return a lot of data - probably a frame of video
endpoint.read(98304)
```

```
# This ends the transfer and switches off the LED.
device.ctrl_transfer(0x01, 0x0b, 0x0000, 0x0001, [])
```

The code above works, which means we successfully reverse-engineered the webcam protocol.

2.6 - Attempting to exploit the protocol

One of our hopes while reverse-engineering the protocol was to find an explicit “LED on” control command sent to the device, as reported by several security researchers in the past.[4] Sadly, this was not the case for our particular setup -- as one can clearly see in the section above, the LED turns on whenever camera data is requested. Having discarded that option, we explored an alternative approach. In the code snippet above, there is a command marked with (*) -- this is the control command sent to prepare the camera for video transfer. We tried replacing 0x0005 and 0x0001 (which are the wValue and wIndex parameters of that request) with different values and seeing if that affects the camera behavior. We reasoned that, in light of the recent revelations about spying programs, there might be an explicit backdoor in the camera that allows an adversary to start it in no-LED mode. We determined by trial and error that any non-zero wValue, and any wIndex with two least significant bits set to 0x01, will result in a video transfer; no other pairs of values result in a video transfer, but all of them turn on the LED.

2.7 - Summary of USB exploitation

Although we managed to successfully reverse-engineer the protocol, we did not manage to exploit it. However, we by no means exhausted all the options -- see Section 5 for possible future work.

[4] "Errata Security: How to disable webcam light on Windows." Accessed: 14 May 2014 <http://blog.erratasec.com/2013/12/how-to-disable-webcam-light-on-windows.html>

3 - Diverting the User's Attention Away From the Camera LED

As the computer science approach to covert surveillance was met with limited success, we changed tack to a cognitive-science-based approach: instead of disabling the camera LED, we worked to simply divert the user's attention away so that they do not notice the light. Specifically, we wanted to measure the capability of the camera LED to cause explicit attentional capture[5] in the user -- make them notice that the light turns on when they were not expecting it to do so. Our goal, if we found that many embedded cameras have poor attentional capture properties, was to attempt to construct a reliable way to induce inattentional blindness[6] in the user -- a state where unexpected events fail to capture the attention of the user, even though the user may normally be aware of and responsive to such events.

A paramount objective here was the ability to quickly iterate and easily test new approaches on different hardware. For this reason, we chose to write this part of the project in Visual C# on the .Net framework, as accessing the camera, capturing images and video, and even broadcasting the captured data were possible in only a few tens of lines of code via the Microsoft Expression Encoder 4 API.

To test the camera LED's attentional capture, we created an application which, when started, would wait a random time interval (up to a minute) and then turn on the camera and take a picture. Members of our team used this application to test attentional capture by starting it, and then continuing to use their device as usual. As there is some lag time between the time the camera is turned on (the LED lighting up) and the time when the first image frame is received by the application, we found that the LED light stays on for approximately 1 second across our devices -- a 2012 HP Pavillion laptop, a 2012 Macbook Pro running Windows via Bootcamp, and a 2013 Surface Pro tablet. Nonetheless, we found that only the tablet's light was bright enough -- the light on the two laptops was not noticeable enough to draw attention away from normal use of the devices. This suggested that placing cognitive load on the user can help induce inattentional blindness on some devices.

In a separate test, we used the same application described above to measure the importance of gaze location on the detection rate of the camera LED. In this experiment, a member of our team would start the application and then keep their eyes focused on a particular spot on the screen for the duration of the experiment. By keeping track of how often we could notice the light turn on as we moved the selected screen location closer to or farther away from the camera LED, we discovered that detection rates diminish significantly when the focus of attention is 5 inches or more away from the camera in any direction during normal laptop use. With cameras mounted above the laptop screen, and when the focus of attention was in one of the lower corners of a

[5] Daniel J. Simons. 2000. Attentional capture and inattentional blindness. *Trends in Cognitive Sciences*, Vol. 4, Issue 4 (April 2000). DOI: http://dx.doi.org/10.1016/S1364-6613(00)01455-8

[6] *Ibid*.

15” laptop screen, the camera LED was essentially unnoticeable. This suggested that active management of the user’s area of attention focus may help induce inattentional blindness.

Having found evidence for two methods that can cause inattentional blindness in the user, we decided to test the predictive power of our ideas. We made the prediction that the camera LED activity will not be noticed by a user who is watching an engaging video (placing them under constant cognitive load) and is forced to respond to application pop-up messages that appear in the bottom-right corner of the screen at the same time as the camera LED flashes (area of focus management).

In order to check the validity of our predictions, we created a formal experiment and tested 14 participants across the same three devices as before: an HP Pavillion laptop, a Macbook Pro and a Surface Pro tablet. Test subjects were told they were participating in a behavioral study, and they were asked to watch an interesting YouTube video describing the science of hockey slapshots.[7] The video was played in the browser, in the YouTube wide player format as shown in Figure 4. We chose this video specifically because many of our friends and most likely test subjects play hockey and we felt they would be fascinated enough by the slow-motion footage in the video that this would help induce inattentional blindness. As part of the experiment, we also designed an application that would pop up a window in the bottom-right corner of the screen, with a randomly-generated task such as “please press the X button,” or “please move the mouse any amount upward,” as shown in Figure 5. Test subjects were briefed to expect these tasks, but were not told where on the screen they would appear, how often or how many times; no further information was given to the test subjects.

[7] Destin Sandlin, Cold Hard Science: Slapshot Physics in Slow Motion - Smarter Every Day 112, Accessed: 14 May 2014 <https://www.youtube.com/watch?v=IsCdywftyok>

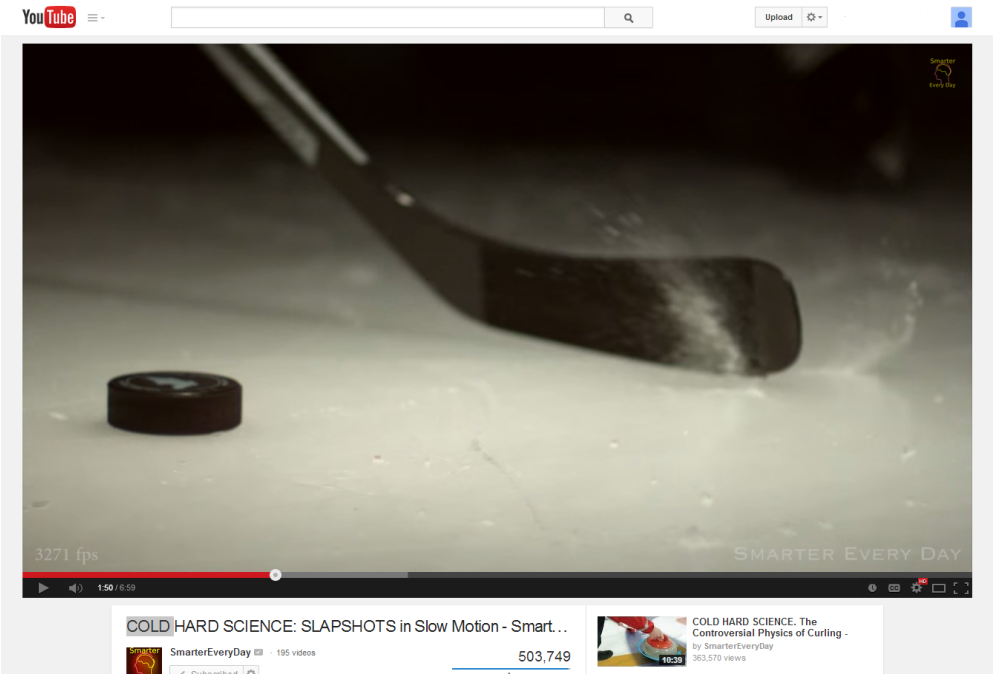


Figure 4: YouTube video as used in the experiment.

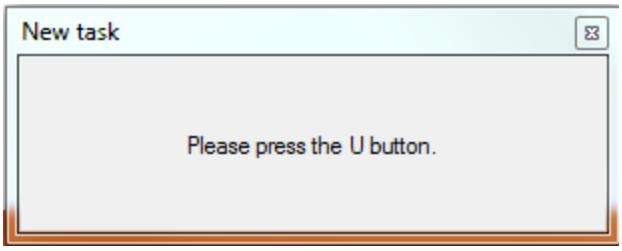
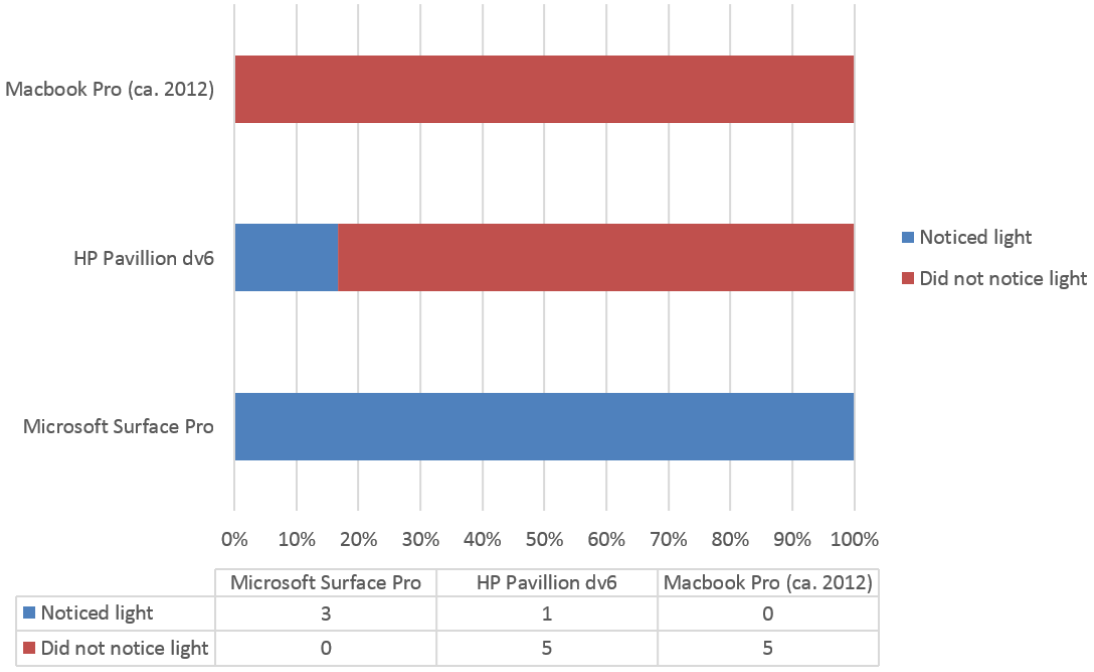


Figure 5: Pop-up window appearing in the bottom-right corner of the screen during the experiment.

The experiment application launched pop-up windows 20 times during the video, in intervals of 17-23 seconds apart chosen uniformly at random, and each remaining open for 4 seconds. During odd-numbered pop-up windows, the application also triggered the camera on the device, taking a picture and therefore leaving the camera LED on for approximately one second. After the test, subjects were debriefed by asking them open-ended questions such as “How did that go?” and “Did you notice anything out of the ordinary?”

We present the experiment results on Figure 6. The results strongly confirm our hypotheses, as no Macbook Pro user, and only one out of six HP laptop users managed to notice the camera LED flash during the experiment. Anecdotally, the one HP laptop user that did notice the camera light turn on revealed during the post-experiment interview that she has no interest in hockey and did not find the video engaging. The experiment shows that the Surface Pro’s camera light may be too strong, and the screen too small, for our distraction techniques to work.



	Microsoft Surface Pro	HP Pavillion dv6	Macbook Pro (ca. 2012)
Noticed light	3	1	0
Did not notice light	0	5	5

Figure 6: User attention diversion experiment results.

From our work, it is easy to see that a combination of constant cognitive load and active management of the user's area of focus can make a user fail to notice the camera LED flash on some devices. Between the HP Pavillion and the Macbook Pro, both of which are 15"-class laptops, we were able to demonstrate a technique that can be used to covertly capture images from an on-board camera with a 90.9% success rate. Our technique could easily be repurposed for remote use by logging the user's work habits, capturing images during high cognitive load situations while displaying a notification in a corner of the screen, and sending the recorded images to a remote server. All of these modifications are trivially implementable in a few hundred lines of C# code on the .Net framework with Microsoft Expression Encoder 4. Since using the technique in this setup does not require any elevated permissions aside from a simple firewall exception (which network-capable applications commonly install), we feel that this is a serious design flaw that can only be rectified by making camera LEDs more powerful or adding different OS-level audio-visual cues that the camera is in use.

4 - Covert Image Capture on Android

Several observations and trends made Android phones the next most interesting target for our investigation in covert camera surveillance. In 2013, about 1.5 billion people on the planet used smartphones.[8] According to Strategy Analytics, smartphone sales reached 990 million in 2013 with about 40% year-over-year growth, which is slowing down for no reason other than their high level of penetration in the American market.[9] These devices are becoming an indispensable part of users' lives. According to the Kleiner Perkins Internet Trends report, the average user reaches for her smartphone about 150 times each day.

The fact that users carry their smartphones around virtually everywhere they go makes these devices prime targets as bugs for recording sound and images. Other factors, such as the typical presence of cameras on both sides of these devices, as well as the absence of LED indicators that notify the user of camera activity the way laptop cameras do, make smartphone-based surveillance especially concerning. These facts, combined with recent revelations about large-scale surveillance programs, make the question of whether such exploits are possible even more interesting.

Our focus was on Android phones for two reasons: Firstly, smartphones running the Android OS accounted for more than half of US smartphone sales[10] and, according to Strategy Analytics, 78% of those worldwide in Q4 of 2013. Furthermore, the more open nature of the Android operating system and its looser security features and app approval process, compared to its rival, iOS, make it an interesting target. Specifically, the fact that in-app access to the camera on iOS is much more strictly secured that on Android is the concern that led us to explore the possibilities.

What we found was surprising: apps running in the background on Android are capable of taking pictures and recording sound with no indication to the user, even when the phone is locked. In other words, even after being closed by the user, an Android app could be capturing sound and images continuously and uploading them to a remote server while the phone is active or locked. To achieve this, the app runs in background mode and attaches a preview to the screen that is small enough to be practically unnoticeable by the user.

We created an application to demonstrate this capability -- it runs a background service that takes images and sends them to a server, which then serves the recorded data through a web interface. If the user checks their running apps, they will not see the app running, as shown on Figure 7:

---

[8] "2013 Internet Trends" Accessed: May 10, 2014 <http://www.kpcb.com/insights/2013-internet-trends>

[9] "Android captures 79 percent of gloabl smartphone shipments in 2013" Accessed: May 10, 2014 <https://www.strategyanalytics.com/default.aspx?mod=reportabstractviewer&a0=9318>

[10] "Who's winning, iOS or Android? All the numbers in one place" Accessed: May 10, 2014. <http://techland.time.com/2013/04/16/ios-vs-android/>

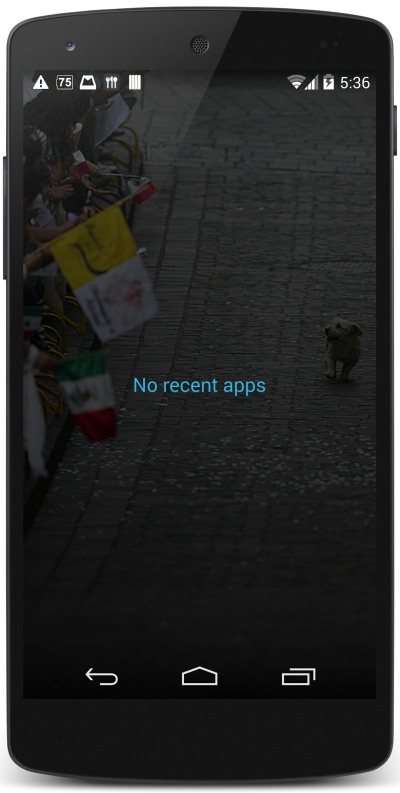


Figure 7: Our android app will not appear on the recent apps screen.

The app, however, sends its data to a remote server that we set up, as shown on Figure 8. The data collection can be configured in real time through a web interface. The captured images, as well as a series of other statistics, including battery life, connection type, user ID and GPS location, are displayed. In our app, the GPS data is collected from the Android cache, which does not cause the OS to display the GPS indicator in the status bar, as this might attract attention to the app's suspicious activity. This makes the data slightly stale, but works very well in practice as the cache is updated anytime any app requests up-to-date location data, which empirically is once every few minutes.

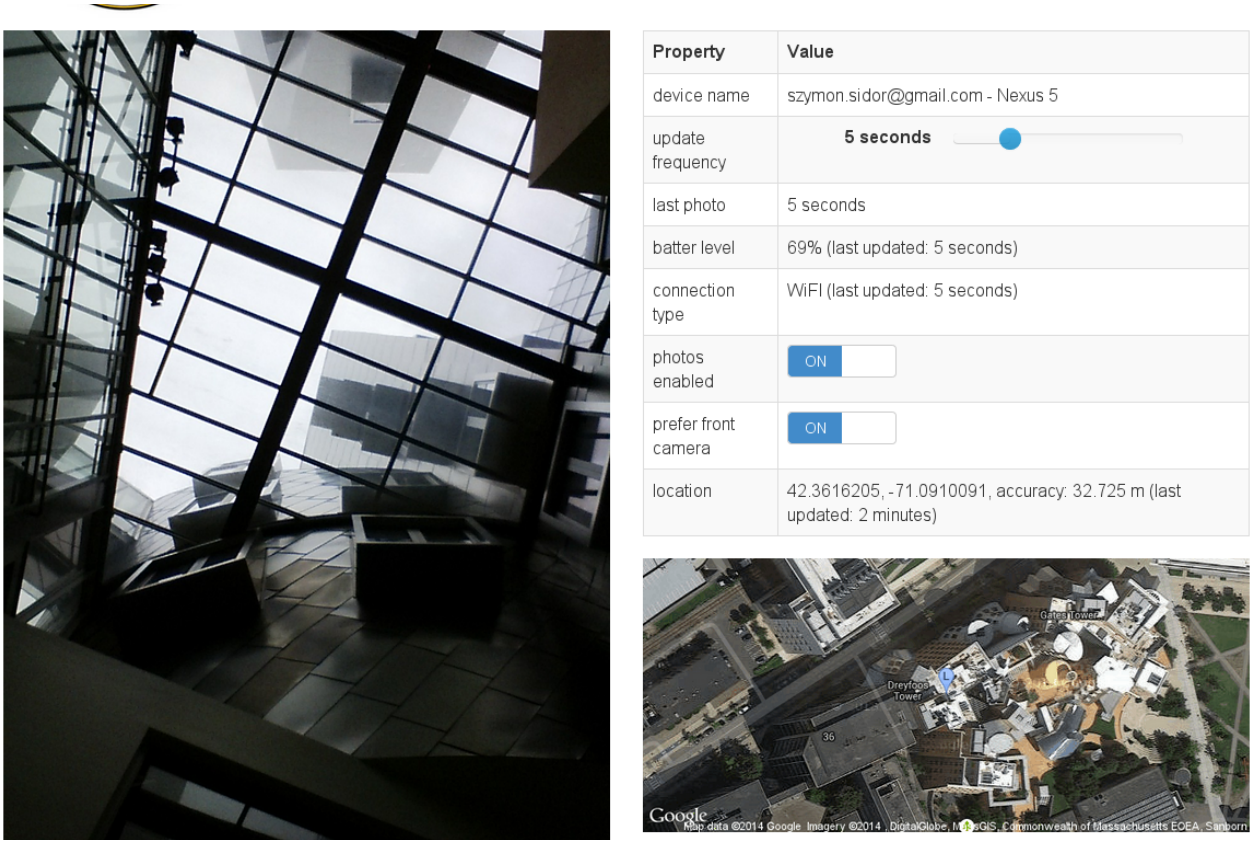


Figure 8: Screenshot of our app's web interface (the attacker's view), with the hacked smartphone's location, camera output, and other information.

4.1 - Technical Overview

This overview, which assumes familiarity with the Android SDK, explains how the task described above can be easily implemented and shipped with any Android application. According to the Android developer documentation:

> Taking a picture usually requires that your users see a preview of their subject before clicking the shutter. To do so, you can use a SurfaceView to draw previews of what the camera sensor is picking up.

Essentially, taking an image using the camera requires creating a Preview Class containing a SurfaceView and passing it to a Camera object. The Preview must implement the android.view.SurfaceHolder.Callback interface, which is used to transmit data from the camera hardware to the app. We must then use the Camera.startPreview() method to start updating the preview before any images can be taken using Camera.takePicture(). This is Android's way of making sure that pictures are only taken when the user sees the camera feed on-screen. Our first attempt was to use a background service to feed the Preview to a View that

isn't attached to any Activity, while having no apps open. This simply fails with the following message: `RuntimeException: takePicture failed.`

Our next approach was to resort to what Facebook does to display its well-known Chat Heads. An app, which runs as a background service, can attach a Preview to the screen, and use that preview to take a picture. Of course, this can't be a problem since the user sees the Preview and would suspect that the application is capturing their photo. Our first attempt at hiding the Preview was to make it invisible or transparent. However, Android ignores these properties for a Preview. Our second try was covering the Preview with another View object, but doing so still gives the user a cue about suspicious activity. The interesting fact, which we then discovered was that a Preview could be resized to an arbitrarily small size. Our last and most successful approach was to resize the Preview to a 1x1 pixel square. In this case, the Preview is in fact on the screen, but is virtually impossible for a user to notice as most contemporary smartphones ship with high-resolution displays. An even more surprising fact is that the camera can continue to take pictures even when the phone is locked and the screen is off.

4.2 - Mitigating The Problem

Some may argue that what we describe as exploiting a vulnerability is only a feature that allows for more flexible applications that enable the use of the camera without requiring real-estate on the screen. However, we strongly believe that the absence of better enforcement of notifications during camera use crosses the line between flexibility and risk to privacy. The ability of a smartphone to continuously and surreptitiously capture and upload sound and video, combined with the loosening of the application approval process on the Play Store to allow for scalability, enables massive surveillance, and can seriously compromise the privacy of users in bulk.

There are several paths to mitigating this problem. For users, the only real solution is to be wary of suspicious application activity, and to pay attention to the permissions that applications ask for. For instance, an application that has no use for the camera or microphone should not be given permission to access them. Furthermore, users should make every effort to keep their Google accounts secure, perhaps using two-step authentication, because it is possible to remotely and silently install applications on an Android phone using nothing but the corresponding Google account. Moreover, users should uninstall unused apps, and look for suspicious activity by inspecting the battery and bandwidth usage of each application. Looking at background services can of course also be useful, as most apps do not need to be running services in the background.

The real fix, however, needs to come in the form of a better implementation of the way users are notified of camera and microphone use. Taking hints from iOS, Android should properly notify the user if the camera, microphone, or GPS is being used by displaying a persistent clearly visible indicator, perhaps in the status bar. This should come in addition to an enforced minimum size for the camera preview which is much greater than 1x1 pixel, and visual indication that the camera or microphone are still on when the phone is locked but the screen is still on.

5 - Future Work

We approached this general problem in many ways but none of them exhaustively. Here are possible extensions of the project which we did not manage to tackle because of the time constraint.

USB exploitation

This part of the project was mostly focused on understanding USB protocols -- how can they be reverse engineered, replayed and possibly exploited. Having built the basic understanding, there are many ways it can be utilized:

- **trying out different devices -** all the ideas presented are pretty general and we expect that having reverse engineered one camera it is much easier to reverse engineer more models. It requires access to the cameras though.
- **trying out more commands to locate backdoor** - we only tried out few hundred commands mostly due to time constraint and the fact that human is required to inspect whether the LED turned on or not. We envision a setup where a photoresistor is connected to a Raspberry Pi GPIO, which has ssh connection open to the laptop that enumerates different commands. This would allow us to test hundreds of thousands of commands overnight.
- **invasive methods** - all our approaches so far treat the camera as a black box. We could instead try opening it up and learning more about the internals to help understand its principle of operation better. In particular one of the ways that we believe would prove successful is to remove the chip inside the case and try to load the firmware from that chip. Inspecting the firmware would immediately reveal any backdoors, should such backdoors exist.

Diverting the User's Attention Away From the Camera LED

While our experiments in this part of the project conclusively demonstrated design deficiencies in several hardware products, we have barely scratched the surface in terms of the number of devices with built-in cameras we tested. Future work could evaluate a much larger collection of laptops and tablets across many different manufacturers for similar issues.

Furthermore, all test subjects in our experiments were college students between the ages of 17 and 24, and it would be interesting to see if people in other age ranges have a better or worse chance of noticing the camera LED. It would also be interesting to change the types or amounts of cognitive load test subjects are under when the camera LED flashes, to see to what extent this affects detection rates.

Finally, a possible follow-up to this project would be an attempt to build a (possibly privileged) computer application that tracks whether and which apps are using on-board sensors like the

camera or microphones. Such an app would then be able to notify the user using supplementary techniques, such as popping up notifications on the screen or playing audio cues. Alternatively, the app could constrain the use of sensors to a white-listed set of applications explicitly approved by the user.

6 - Conclusion

The goal for this project was to discover and evaluate possible methods for covertly surveilling user activities using compromised PCs and smartphones. This was split into three primary efforts, the first of which used USB packet sniffing and injection to discover and exploit backdoors in embedded cameras in laptops. The second effort conducted a user study to determine the effectiveness of indicator lights in alerting users to camera activation on their PCs. The third effort investigated possible ways for an Android application to capture and send video to an attacker without alerting the user.

These efforts yielded two contributions to the field. First, our user study demonstrates that camera indicator lights, as they are currently designed and implemented, are terrible at indicating to users that their camera is active if the user is distracted or not expecting that indication. Second, we found that the current Android application model allows for an application to give practically zero indication to the user that the camera is recording -- a clear vulnerability in our eyes.

This leads us to a single conclusion: covert surveillance on the PC and Android is surprisingly feasible, and deserves much more attention if such surveillance is to be prevented in the future.