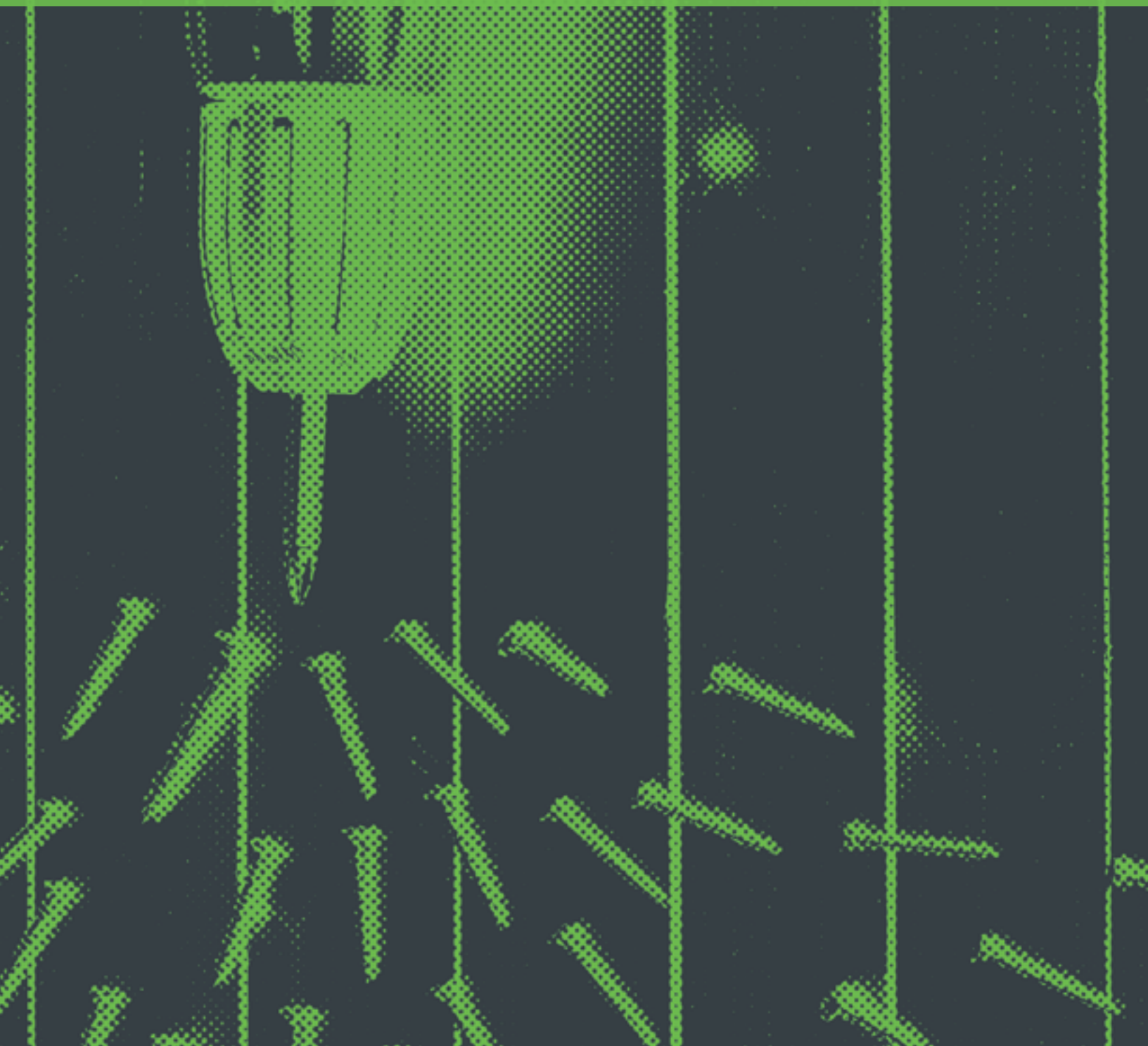
Duo Labs Presents

Bug Hunting

Drilling Into the Internet of Things (IoT)

DUO LABS

CISCO Duo Security is now part of Cisco.



This page intentionally left blank.

AUTHOR
MARK LOVELESS

DESIGNER
CHELSEA LEWIS

PRODUCER
PETER BAKER

The TL;DR

We try to come up with a way to speed up the process of evaluating IoT devices, then pick an IoT device to try the process out on. The process we developed, while software-focused, yielded results fairly quickly on our target app:

- **Static passwords used for updates to a master database located on the vendor website were hard-coded into the accompanying IoT app.**
- **The IoT device in question was expensive, but could be readily identified by a potential thief remotely via Bluetooth scanning.**
- **GPS data used for inventory tracking in the event of lost or stolen devices could be forged.**

A lot of security measures were in fact done correctly, and much of what one might call “low hanging fruit” was protected, which was impressive.

Background

The Internet of Things, or IoT as all the cool kids say, is always in the news as of late. There are a number of these new devices that are useful and have become a “must-have” item, but many of these new items seem to have been given an IP address or Bluetooth chip simply to sell to a market willing to have the latest tech - regardless of how useless the device seems. Sometimes the introduction of “smart” to a device makes sense and changes the landscape completely - your cellphone becoming “smart” and talking to the internet is arguably the best example of this. Products like a smart toaster? Not so much.

There is a danger in approaching IoT from a security research perspective - mainly that it is nothing more than **stunt hacking** and not worthy of researchers’ time and effort. Unfortunately as these devices become commonplace, there are areas of concern for the average consumer that validate the research - especially if a device has security elements or utilizes personal user data.

While this was originally going to be a report on a particular smart device, it has since turned into a report on how to approach this brave new world of IoT from more of a software perspective (as opposed to more of a **hardware perspective**), and we’ll simply use a smart device as a guinea pig to show how it is done. As a researcher, I can safely state I am somewhat lazy, as I’d prefer to simply **hoodie up** and start shell gathering. So I wanted to come up with a process that was quick and thorough for examining IoT technology.

The Best Place to Look

Where does one look for these IoT bugs? It seems like the device itself is the most obvious one, but it is usually not the case when you are looking for the most impactful bug. For example, if you know the type of chipset in the IoT device, then you know what the device is capable of doing and this should guide the investigation to a degree. But honestly, the best place to look is the associated app that will reside on the phone. Most IoT devices rely on the owner's phone to act as a router to the cloud, so all information typically passes through and can be examined in one way or another.

In the future, many IoT devices will not be as reliant on moving data via Bluetooth to a phone. As battery life and transmission of data improves, IoT devices will be moving data via Wi-Fi or even LTE (Long-Term Evolution, the current standard most commonly used by telecommunication companies to move voice and data via smart phones over their networks). But until then, the phone is the best way to examine communication between IoT device and cloud.

The Digging

So what does one look for? Here are the items currently on the checklist:

- Making sure the **basics are covered** – encrypted communications, certificate pinning, secure Bluetooth pairing, storage of data on phone/cloud secure, no hardcoded passwords, etc.
- Ensuring encryption used is strong, not susceptible to easy cracking, replay, etc. in the case of SSL, strong cipher suites, etc.
- Determining no unintended information leak. This includes excessive permissions, or unintended data in uploads.
- Verifying the application is not using out-of-date libraries or SDKs that have known security issues.
- Checking that security elements perform as designed and cannot be bypassed.
- Ensuring that values or parameters cannot be changed to cause issues during use of the application or IoT device.
- Designing real-world attack scenarios, then trying to connect the dots to see if they can be implemented using a combination of flaws found and existing limitations (or lack thereof) in the technology two-factor involved.

We will do Bluetooth scanning for the device and gather details remotely, perform static analysis of the app that's on the phone, look at whatever data the app stores and what else it accesses. We will examine traffic between app and device, as well as app and cloud. During the process we will develop real-world attack scenarios and try to either prove they are real, or show how they will not work. The result should be a fairly clear picture of exactly how safe (or unsafe) this device is.

Like most researchers we will not just start at the top and move to the bottom of our list, we will perform most of the items in parallel, and will most likely encounter moments where one issue will cause us to jump between steps to help get the full picture. My personal preference is to set up monitoring between device and app as well as between app and internet first, then start using the app and device as intended to start gathering data. I jump between app, device, monitoring, and source code. I also Google the hell out of anything I don't understand or that looks odd.

Our Example Target

The target we are going to look at is the **Milwaukee ONE-KEY M18 Fuel 1/2" Drill/ Driver**, from Milwaukee Tool. This is a cordless, brushless drill from a reputable and somewhat high-end tool company. **ONE-KEY** is the “smart” part of the tool, and is something available on a number of different Milwaukee Tool products.

Before you eyeroll too much, there are some actual interesting things you can do, at least per the website and various videos from Milwaukee Tool. If you have multiple smart tools, you can track all of your inventory via the app (and the Milwaukee Tool website). If you are a contractor with a large inventory of tools, that has an appeal. Anyone else with a ONE-KEY app on their Android or iPhone will scan for tools in the area constantly, and discretely phone home the GPS coordinates of any tool encountered to the Milwaukee Tool website, so if your tool walks away and ends up somewhere else and is scanned by someone else, you can locate a missing or even a stolen tool.

The drill. When the robots rise up, they’ll use these as weapons.

You can remotely disable the tool from working in case it is stolen. You can build multiple profiles for each tool in the app on your phone, each with different settings (speed, torque, etc) for different tasks, and then push the profiles to the drill. For example, if there are complaints that some of your crew are over-tightening wood screws and causing issues for others on a job site, it can be addressed by updating and pushing out a profile that adjusts the torque. You can add notes to the tool so you know who checked it out and other handy information.

Bluetooth is active, four tool presets to edit/upload via your phone.

Now maybe this stuff isn't interesting to you the computer nerd, but as you can see from a just a few highlights above, maybe this smart tool thing has a point. And frankly, from just a regular "tool" standpoint, this is one hell of a nice drill. Even without the ONE-KEY stuff, this is considered a premium tool and is easily 4-5 times more expensive than the drill I have. In fact, this is the nicest power tool I've ever used. But the ONE-KEY part opens up some interesting possibilities for us to explore.

Getting Started

After making sure everything worked as designed between app, drill, and the internet, the latest APK was downloaded for the Milwaukee ONE-KEY application (note: the main version looked at during this process in Jan/Feb 2017 was 3.0.2, although a few other versions after that were glanced at, most changes to those versions were to the GUI). Reversing Android apps is a lot easier and I used a Nexus 5 specifically for experimentation, but usually run the iOS version on my iPhone 6 so there is a pristine setup in case I need to check against an unaltered version, particularly for regular app usage.

For Bluetooth traffic, I use a combination of different tools. I have several USB devices, including an **SMK-Link Nano Dongle Bluetooth 4.0**, a **Sena UD-100** (with **different antennas** that extend the range up to half a mile), and an **Ubertooth One** that have all been tested to work with my Ubuntu 16.04 desktop system.

There are a variety of command line tools on Linux I use, such as **hcitool**, **btscanner**, and the various command line tools that work with the Ubertooth (all of the **ubertooth-*** programs).

I also use a couple of apps - **LightBlue Explorer** from **Punch Through**, and **nRF Connect** from **Nordic Semiconductor**. The reason for the amount of monitoring tech is because monitoring Bluetooth - especially sniffing - is extremely hard to do accurately. Most presentations involving Bluetooth that one sees at various security and hacker conferences all have the "sniffing Bluetooth is hard" slide in the deck. And many of the tools do not give the same results so you end up combining output from multiple sources to get a complete picture.

Since the advent of 4.0, the whole Bluetooth Low Energy (BLE) standard was integrated with Bluetooth Classic and it is just considered plain old Bluetooth now. Not all of the sniffing tools look at the **Bluetooth 4.2 spec** the same (4.2 is the latest spec with wide adoption as of this writing), and since the authors of each tool had different needs, they may not return the same data elements you might be looking for. This is further complicated by weird implementations by the IoT vendor and so on.

For network traffic heading to the internet, normally a sniffer such as **Wireshark** is more than fine. However, most traffic from applications these days is done over SSL, so extra steps are needed for sniffing. Probably the easiest method for monitoring is to use **mitmproxy**. Using an old laptop running Ubuntu with an Ethernet connection as well as configuring the wireless interface to run in ad-hoc mode advertising an access point, one can run **mitmproxy in transparent mode** and capture the encrypted traffic.

If the application uses certificate pinning, there are more steps to be performed to bypass it for testing purposes - basically by decompiling the APK, tweaking the code that checks the pinning to return nulls, recompiling, and signing the app with your own key. For details on these steps, refer to Appendix A below.

For static analysis, one tool that comes in handy is the open-sourced **apktool** running on Linux. It is great for a quick decompile and some quick searching for low-hanging fruit. It is also a key tool for recompiling the APK after you've disabled pinning. Of course no Linux workbench should be without **adb** for having a look at the Android-based phone. The one commercial tool I am using right now is **JEB**. Decompiling into Java makes things much easier to read than reading smali, and it is feature-filled enough that most people won't require much of anything else.

The App

It helps to explore the app used to talk to the IoT device and the cloud, just to see what it is capable of and to help plot out a course of action. It is easier to obtain and work with application source code by grabbing the Android version, so after grabbing the latest APK, apktool was able to quickly examine the disassembled Milwaukee ONE-KEY app in **smali format** (this is out of laziness, I started a full disassembly in JEB but that takes a bit and I wanted to get a quick peek).

When an app is developed, most developers don't design from the ground up, they use libraries to make the process go easier. The first thing to do is look to see what libraries are used and to check their versions to ensure they are running a library that does not include a security issue. The **Flurry** SDK is used to track app usage and report it back to the app developers. While previous versions of Flurry had security issues, the ONE-KEY app uses a later version. The same for **OkHttp** - an earlier version had an issue with the handling of certificate pinning - but ONE-KEY is using the latest version of that library. In fact, all of the libraries used by ONE-KEY were the latest at the time of review.

A note on library version checking - some libraries include it in their source in some form or fashion. Others are not so obvious, you have to figure out what is a recent feature or fix that was added to the SDK, and look for that feature or fix.

In regards to certificate pinning, it helps to search through looking for various things pertaining to pinning, such as "X509" and "TrustManager," to find the pinning stuff quickly. Sure enough, they are using the OkHttp library for pretty much all of their client internet functions, including SSL connectivity and certificate pinning. The full disassemble with JEB to see it all in Java made it quite readable.

All in all, this was fairly impressive. Use of SSL exclusively for communications, certificate pinning, latest libraries - but then there is the bad.

There are base64-encoded credentials in the app for talking to the Milwaukee Tool web presence. This more or less undoes a lot of the good. There are some mitigating factors we will discuss later, in that these factors imposed some limits. I did not use these credentials, even for just exploring and poking around (because it is 2017, not 2001, as well as the whole felony aspect), but as they are used to establish authentication for the app itself and to get a session token for all transactions, they become the keys to the kingdom - master credentials as it were.

User accounts are tracked separately (via email and password) but one of the things ONE-KEY does is offer GPS tracking of ONE-KEY tools via the app - even if the tool does not belong to you. For example, if the ONE-KEY app is installed on someone's smartphone and they walk by a job site with a Milwaukee ONE-KEY tool, the app will "see" the tool and phone home with GPS coordinates stating its location. To update the tool's entry in the main database with the GPS location and a timestamp requires credentials. While the passerby's user account is not associated with this tool on the job site, it is not in their "inventory" so the user credentials can't alter the data on a tool they don't own - but this master ID and password statically stored in the app can.

Data used by the app is stored in a database called mke.db, and while its data can be rewritten offline and then uploaded back to the phone, there was no method to allow for illicit data values to find their way onto the tool (e.g. adjusting the RPM to unsafe levels). To push illicit data to the app required pulling it into the application, and this cleaned up the altered data. Pushing it directly seemed like it could be a possibility via Bluetooth directly, however the difficulties in recording the Bluetooth packets to see how this is done (see the Bluetooth section) prevented designing a test for this possibility.

A user of the app can create a "guest" account with a password, and assign a few different rights to it including editing of tool controls, locking and unlocking of tools, and viewing and editing of inventory. These controls are enforced against tools via queries to the master database with the identity of the tool in question, and checking against what the owner of the tool has allowed.

Bluetooth

The Bluetooth electronics for the drill are buried in the base, although there is a coin cell for Bluetooth power when the RedLithium battery is not attached.

Coin cell is behind a small panel that is underneath where the RedLithium battery attaches

As suspected, a normal hcitool scan did not show the drill, but running hcitool with the lescan option for Bluetooth Low Energy (BLE) devices did, after a some trial and error by turning off other nearby Bluetooth devices, the drill's MAC address was obtained. Armed with that I was able get a little more info:

```
thegnome@fang:~/Projects/milwaukee$ hcitool leinfo 00:07:80:CF:76:BC
Requesting information ...
        Handle: 75 (0x004b)
        LMP Version: 4.0 (0x6) LMP Subversion: 0x3
        Manufacturer: Bluegiga (71)
        Features: 0x01 0x00 0x00 0x00 0x00 0x00 0x00 0x00
thegnome@fang:~/Projects/milwaukee$
```

Bluegiga itself has no known vendor-specific vulnerabilities that I could find from Googling, and the “71” on the manufacturer line is most likely a build version. This particular build version places it at about a 2013 release year, and although this suggests an older version, it is considered a stable one by most developers. A probe from LightBlue Explorer also helped, with the model number string BLE112 referring to a specific **Bluegiga product**:

Light Explorer screen grab, note the BLE112

The important thing to note is that this tells us the tool is mainly speaking BLE, and does not use features from the full Bluetooth 4.2 spec which include a number of security features. This suggests a possibility for flaws involving the BLE traffic itself between app and drill. For example, there is no security during the pairing process, it uses the method known as “Just Works.” There is no security code to look up, it just does the pairing (technically there is a security code, it is 000000, but the tool offers no method to display a code so this is what is done).

After unsuccessfully trying btscanner, I used ubertooth-btle once I had the MAC address, and tried pairing multiple times as well as uploading a profile while attempting to monitor the traffic between drill and app. As stated, sniffing Bluetooth traffic is a non-trivial task, and after spending the better part of a day I was still dropping a few packets (the same task would give a different packet count on the few times I got anything at all). The command line options I used that gave the best results were as follows:

```
thegnome@fang:~$ ubertooth-util -f -t00:07:80:CF:76:BC -r drill09.pcap
```

The encouraging thing was that it appeared encryption was in use due to a lack of plaintext in some of the packets being exchanged during a profile update, which was a pleasant surprise. The BLE112 supports AES 128 natively, and the source code suggested AES 256, but the mere presence was nice regardless of key size used. To give you an idea of the effort of acquiring those packets, it took 7 pairings before a capture with an initial CONNECT packet was accurately sniffed, even though there was one happening with every pairing. It took a total of 21 captures to get a full pairing session. That was just the pairing - I never got a complete data transfer of a profile update, I am assuming the traffic was encrypted based upon looking at several individual packets from the profile update as there was no plaintext.

There were no major flaws uncovered, with the exception of being able to easily connect to the device using Bluetooth apps since a BLE112 supports up to four simultaneous connections.

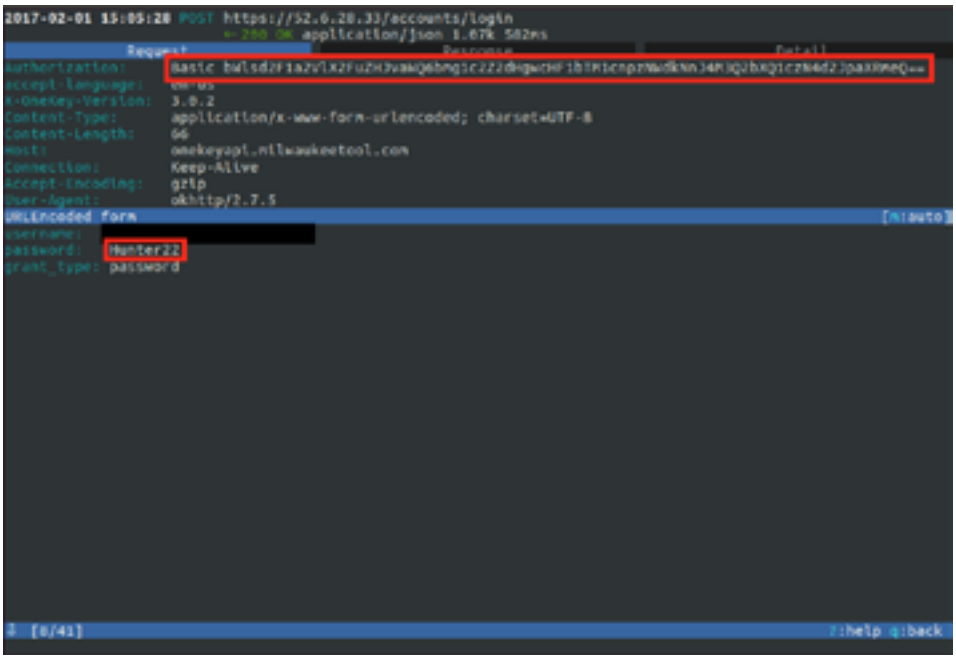
As you can see, Bluetooth examination is entirely dependent upon both the quality of the tools being used, and the amount of time one has to invest in the process. As a researcher, finding Bluetooth implementation flaws between an app and a power tool is not as important as finding flaws in something more critical (e.g. the SSL implementation), and so between working on other projects, (most researchers are usually working on more than one project at any given moment), a researcher has to evaluate how much time they are willing to put into it. At best, I can say for this project that if there is a flaw in the Bluetooth communications, the average attacker with average Bluetooth attack tools will not be able to exploit it.

The Network

The network traffic uses SSL exclusively and certificate pinning was verified to be working as intended. The cipher used by SSL is always ECDHE-RSA-AES128-GCM-SHA256. While not as strong as AES 256, it is still quite robust and certainly considered safe, and it has the advantage of being less CPU intensive. This helps preserve some battery usage for mobile clients (as an example, many Google-written apps use this particular cipher with battery life in mind).

After the SSL handshake, there is a short exchange to identify the version of the app (in case an update is available). Then a **Basic Authentication** is done using the previous-mentioned base-64 encoded credentials, along with the username and password of the end user. If the username and password are valid, an access token in the form of an OAuth 2 bearer token is returned.

While examining the traffic, it became clear that all new inclusions and updates to any data associated with a tool owner and their tools required this bearer token before sending and retrieving data from the website. Having the bearer token is great, although this token has an unusually long time until expiration - 365 days. Normally, a bearer token is set to expire after an hour or two.



Login response. Bearer token, user_id in UUID format. Note the "expires in" value.

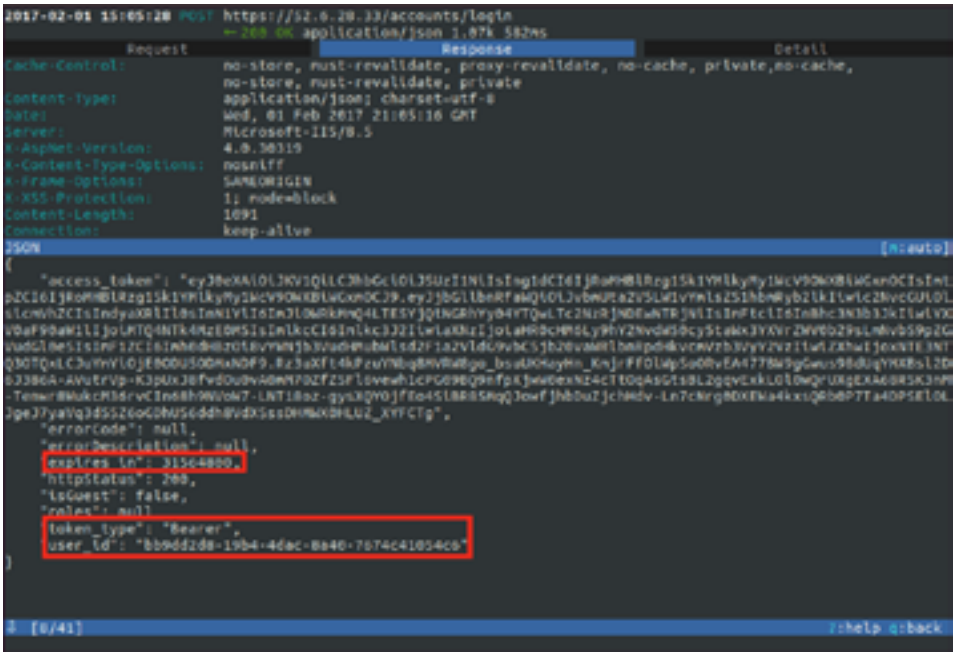
To help track the user during access, when the bearer token is returned, a unique “user_id” value is also returned in the UUID format of xxxxxxxx-xxxx-xxxx-xxxx-xxxxxxxxxxx, assuming the end user’s username and password were correct. Both the user_id and bearer token are used until the app has to login again, when a new bearer token and user_id are assigned. UUIDs used by the ONE-KEY app and by Milwaukee Tool’s website all appear to be RFC 4122 compliant.

```
thegnome@fang:~/Projects/milwaukee$ uuid -d bb9dd2d8-19b4-4dac-8a40-
7674c41054c6
encode: STR:     bb9dd2d8-19b4-4dac-8a40-7674c41054c6
        SIV:     249385102245824204800662671554287523014
decode: variant: DCE 1.1, ISO/IEC 11578:1996
        version: 4 (random data based)
        content: BB:9D:D2:D8:19:B4:0D:AC:0A:40:76:74:C4:10:54:C6
                 (no semantics: random data only)
thegnome@fang:~/Projects/milwaukee$
```

The bearer token stored on the mobile device is a security risk when it has this long of a lifespan, and coupled with the user_id, it gives an attacker who has compromised the phone the ability to act as the user and manipulate such items as inventory.

Most operations require the user_id for them to work, although a few don’t. The main one discovered was the GPS location update of the tool (although this makes sense, since it is required for the whole process of GPS updates from any phone about any tool). And while not quite as important, one can also rewrite the profile data that is stored on the tools themselves by sending illicit updates to the Milwaukee Tool website and allowing the victim user to download them later.

The server returning data to the client always uses X-Content-Type-Options set to “nosniff,” X-Frame-Options set to “SAMEORIGIN,” and X-XSS-Protection set to “1; mode=block.” This is good as it helps prevent a number of security issues:



Login request. End user password requires minimal one uppercase letter and 8 chars long, eliminating the usual “hunter2”.

X-Content-Type-Options: nosniff.

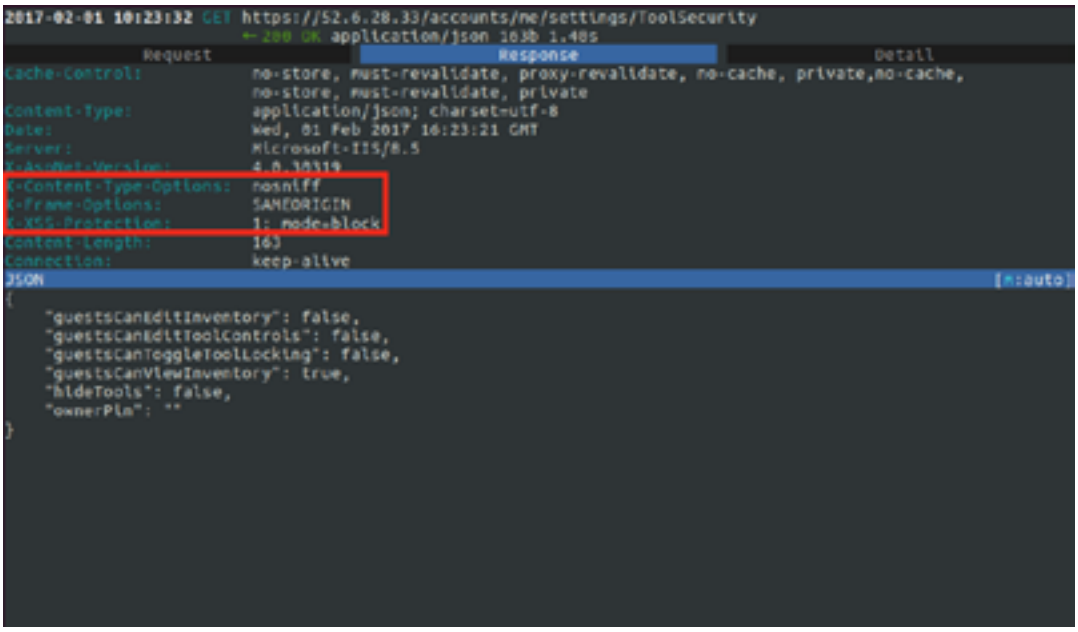
Basically, this tells the browser to not sniff the response buffer in an attempt to identify the MIME type of content being retrieved from a server, and is particularly used in cases where data is being uploaded by users. A few browsers honor this, including Internet Explorer and Chromium. For example, if the server has set the MIME type of the content to “application/pdf,” but the browser scans the buffer causing it to identify the content as something else, the browser overrides the server’s header specifying the content. Nosniff prevents that. To make it clearer, think about it in terms of telling the difference between binary and text data. And if you think there should be no issue with an app on a phone, remember you can also access ONE-KEY data from the ONE-KEY server using a regular browser on a regular computer.

X-Frame-Options: SAMEORIGIN.

This setting enforces iframes from being loaded from other servers, again a decent protection against web trickery when user content is allowed to be uploaded. For example, this will help prevent **clickjacking**.

X-XSS-Protection: 1; mode=block.

This stops web pages from loading that contain **cross-site scripting (XSS)** attacks. The “1” enables detection, and “mode=block” causes the page to not load in the browser.

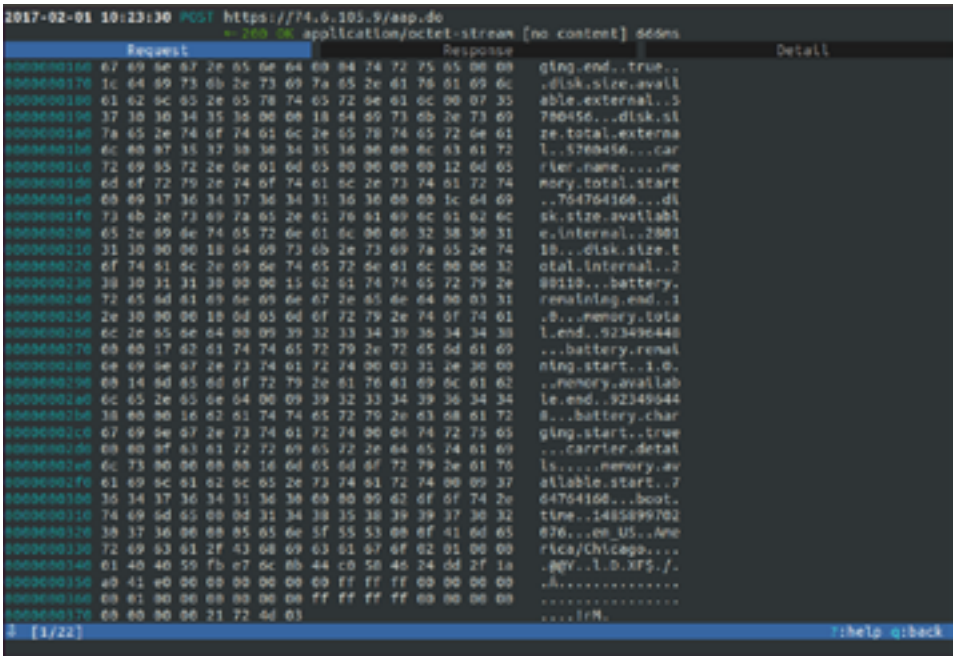


Note the HTTP header responses.

Privacy

The application does collect information on app usage and a bit about the phone the app is loaded onto. Outside of simply analyzing how their app makes its calls back to the website, the ONE-KEY achieves part of this via Flurry, a popular library used to gather app data. None of the data is particularly invasive, which is good.

The app allows you to take pictures and “attach” them to the tool in the database, so the app requires access to your camera. This way, you can take a picture of the tool perhaps in its environment, and you can also attach a photo of a receipt as well - allowing you to know when it goes into service.

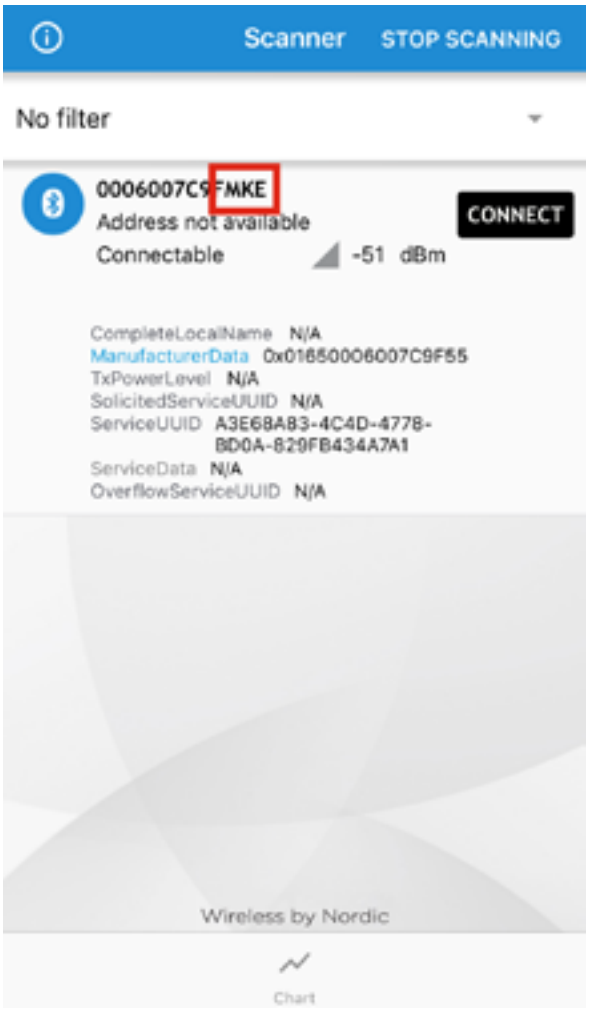


Part of the data Flurry returns. It also reports phone type and build version of the OS.

Since one of the selling points is a “find my power tool” function in the event of loss or theft, the ONE-KEY app gathers GPS data via normal phone APIs like other apps do, such as Google Maps. Some other odd methods in a few other odd apps have been known to use different methods, so it is good to check for this just in case. For example, when we looked at the **Chinese cell phones**, they were using a plaintext web-based location service. The Apple Watch does this as well, or at least the last time I looked - if you use the Maps app while your paired iPhone is off or out of range, it will resort to a plaintext web-based location service, because it doesn’t have the cellular capabilities that the iPhone does.

Most common Bluetooth scanning tools that support BLE will easily detect Milwaukee Tools that have ONE-KEY. There is a setting one can make in the Milwaukee ONE-KEY app that “hides” a tool in your inventory from other app users - preventing another Milwaukee ONE-KEY app user from seeing your tool within their app. This does not restrict the tool from being scanned via Bluetooth scanners, as the hiding feature does not put the power tool in undiscoverable mode.

As the name of the Bluetooth device is hard-coded into the power tool, a quick scan with any number of Bluetooth scanners allows one to fingerprint a Milwaukee Tool with ONE-KEY very easily.



The MKE part kind of gives it away in multiple scanning tools, including Nordic’s nRF Connect app.

While having MKE in the name is enough, looking at the Manufacturer Name String after connecting gives you “MILWAUKEE TOOL”.

Physical Security

When checking an IoT device, it is rare that there are any physical security elements, but in this case, we have a couple.

To push the profiles to the drill, they have to be loaded into the app which would rewrite the levels to proper constrained levels. Additionally, it was impossible to push updates to the drill without explicitly making a physical adjustment on the drill itself:

Normal usage on the left, Bluetooth profile update mode on the right.

This helps prevent a profile update during power tool usage, as one has to manually make the adjustment to allow the update.

It is possible to lock the drill remotely with the ONE-KEY app. This could come in handy in case you want to shut off a drill that has been lost or stolen. Once the drill is locked, it simply does not work (it appears that the power from the battery to the drill's motor is cut).

To turn this off, you need a valid bearer token and a valid user_id associated with the locked drill - not easy without access to a phone that has the latter stored. It might be possible if one could obtain the "guest" credentials for an account (the ONE-KEY app allows you to create a guest account that, by default, can manipulate inventory items), this could allow an attacker to remove the stolen item from inventory after unlocking the drill.

Vulnerability Summary

During the course of this exploration, several flaws were identified. Each of these flaws were either tested for or at least analyzed to see if they were possible. As noted above, several were found, and here is a list of all of them (two have been assigned **CVE** IDs):

Privileged Stored Credentials

The ONE-KEY app includes master credentials in base-64 encoded format that are needed to obtain a bearer token. The bearer token allows for read-write access to information stored in Milwaukee Tool's website. Most operations also require the current user_id value, however not all operations do - the main one that does not require it is the updating of the GPS location of any Milwaukee Tool equipped with ONE-KEY.

CVE: CVE-2017-3214

Recommendation to vendor: Use a more secure method of remote access for the credentials needed for the app.

User remediation: None.

Stored Bearer Tokens with Long Expiration Times

A typical bearer token has an expiration time of 1-2 hours, these have an expiration time of one year, and are stored on the phone for reuse while the phone is logged in. In the event of a compromised phone, it is possible for an attacker to gain access to the bearer token and use it. This also means that the user_id - which is also stored locally - could be used by an attacker with this bearer token to perform user actions.

CVE: CVE-2017-3215

Recommendation to Vendor: Expire the bearer token after a much shorter timeframe, such as 1 or 2 hours.

User Remediation: Log out of the app after each use as a new Bearer token is issued with each login.

Remote Device Fingerprinting

A simple Bluetooth scan from a smartphone with a free app will quickly identify a Milwaukee Tool with ONE-KEY from a distance of 50-100 feet. Using Linux-based tools such as hcitool and the built-in antenna will get the same results, but using a more powerful antenna can allow detection between a quarter and half mile (depending on various conditions).

CVE: None.

Recommendation to vendor: Non-obvious name of device that does not include MKE in the name. While it would be nice to put the tool in non-discovery mode (via the "hiding" feature that prevents other ONE-KEY app users from seeing the tool), this might complicate other features.

User remediation: A tool owner could remove the coin cell battery and disconnect the RedLithium battery. This prevents the ONE-KEY tool's Bluetooth from broadcasting in discoverable mode. The owner could simply hook up the RedLithium only for actual usage, minimizing exposure. On the flip side, this is probably the exact technique a thief would use to help prevent the tool from being found as well.

GPS Reporting Subject to Spoofing

For details, see "GPS exploit" in the Exploit Summary section below. Basically, one can used data obtained from some of the other vulnerabilities to forge GPS queries to make the tool appear to be in a location it is not.

CVE: None.

Recommendation to Vendor: Correcting the other vulnerabilities would prevent this exploit from working.

User Remediation: None.

Exploit Summary

Sometimes the tricky part when finding vulnerabilities is to find ways to practically exploit them. For the application of this technique for IoT devices, one has to think about how the devices are used in the real world. Sometimes a vulnerability can seem devastating or extremely limited, but once applied to the real world and a real device, the opposite might be true.

Tracking Exploit

This is the most obvious exploit found. Having the last three initials in any Bluetooth device as MKE points to it potentially being a power tool, and a quick check after connecting shows the manufacturer to confirm it. A serious thief could simply drive around with their (probably stolen) laptop and $100 worth of Bluetooth interface and antenna, and locate ONE-KEY hardware within a quarter to half mile radius. Using signal strength, the thief could zero in on the location. The thief could then scan for nearby smartphones and fitness trackers (FitBits etc) to see if there are nearby witnesses or perhaps even the owner, and wait for the location to be clear of mobile Bluetooth devices indicating people.

It should be noted that most people who own power tools tend to put them in the same place as all the other tools, like a tool chest or cabinet in the garage or shed at a job site. The thief could then steal not just the Milwaukee Tool running ONE-KEY, but the victim's entire cache of tools. In fact, the thief might just steal several thousand dollars worth of the "dumb" tools, and leave behind all of the ONE-KEYs! A bold thief might return in a couple months after insurance (or the victim's cold hard cash) replaced the stolen items, and have a lot of newer tools to pick from.

GPS Exploit

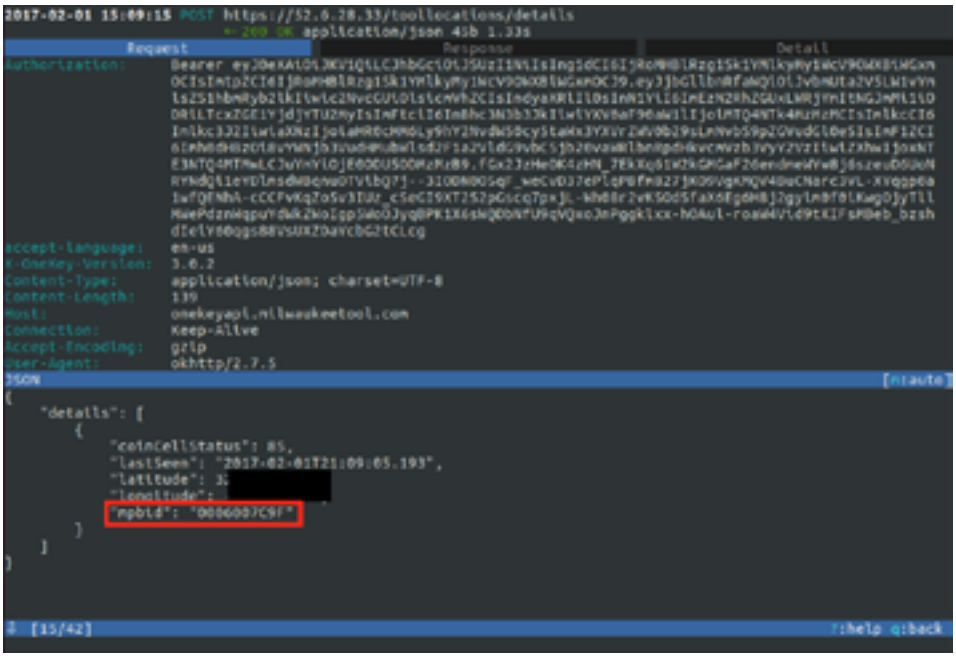
The elements needed for GPS location updates of a ONE-KEY power tool between the ONE-KEY app and Milwaukee Tool web presence require the following:

1. The master user/password from the app.
2. A valid account and password.
3. A valid Bearer token.
4. The tool's unique ID.

All of this takes place under the cover of certificate-pinned SSL with secure cipher choices. But one of the features of the ONE-KEY system is that if the tool is near any ONE-KEY app, even if that app user is not the owner of a tool it detects, it will send a transaction to the Milwaukee Tool website with the GPS coordinates of the tool.

The first item is hard-coded in the ONE-KEY app, as stated in our vulnerability section. The second item requires a working email address, available from a free email service. These are enough to acquire the third item - a bearer token.

The final element is located in the Bluetooth name of the device. In our report, we've mentioned determining the device is a Milwaukee Tool because of the "MKE" in its Bluetooth name, which with our drill is "0006007C9FMKE." The unique ID for this particular drill is the first part - "0006007C9F." Armed with these elements, one can feed false GPS coordinates.



There's our drill, the mpbid value.

To obtain the bearer token, you have to log in, both with a working end user account and password as well as the master user/password. I was not able to confirm that if one logs out with the app that the bearer token is automatically expired, as this requires leaving the world of traffic sniffing and start getting into sketchy territory - actually using the master user/password outside of the app (technically kids, this is code execution on a web server that isn't yours, and is a felony). Now maybe I might push things a bit and whip out the grey hat if we were talking about a device with life and death consequences, but I am not willing to risk a felony while clocked in at work, especially over a power tool.

Adjusting Other Tool Values in the Milwaukee Tool Website

In theory, there may be other transactions besides last known GPS location that can be adjusted. Changes to tool profiles are possible without a user_id, as are acquiring any assigned "guest" account information including the password, although these appear to require additional elements that could not be obtained legally during testing. As there was no attempt made to simply log into the website with these credentials and start poking around, there was no attempt made to look for other transactions that did not require a user_id, except by those observed during regular sniffing.

Vendor Response

For each reported vulnerability listed in the Vulnerability Summary section above, this is the vendor response to each one. The vendor responded to CERT, and this is what was said about each reported vulnerability.

On “Privileged Stored Credentials”

"The "master" credential that was found is only used to identify the type of client to the authentication server so it can determine the type of bearer token to provide. It is used in conjunction with the user's username and password to provide that token. This API Token does not provide a user access to anything else. Also yes, any user or attacker can acquire a bearer token by signing up for a free Milwaukee account. All bearer tokens limit the user to only their data or other publicly accessible data."

On “Stored Bearer Tokens with Long Expiration Times”

"This is correct, we don't expire the bearer tokens because we don't want to force the user to log in frequently. That said, there are circumstances where we do invalidate the token on the server side (e.g. password change, permission changes, etc) which forces the user to log in again and acquire a new bearer token."

On “Remote Device Fingerprinting”

"Understood, I'll need to research this more with our EE team who works on the Bluetooth firmware; my team focuses on the software."

On “GPS Reporting Subject to Spoofing”

"This is also correct and something we are always working to improve. The primary function of the tool tracking feature is to help the user remember which jobsite their tool is currently at or last used at. That said, we do have plans to improve the validation of the client sending updated location information; however, it will not be in the immediate next version released."

Appendix A: Bypassing Certificate Pinning

Certificate pinning prevents performing MITM attacks against HTTPS traffic. In many cases, you can use mitmproxy (mitmproxy.org) to sniff HTTPS, but when certificate pinning is set up properly, you cannot. However, with a few modifications to an Android version of the app, you can adjust the logic to ensure certificate pinning is bypassed. Here are the steps to do this.

0. Prep

We will assume you have access to a Linux system, Ubuntu is a popular distro around Duo Labs. You will need to make sure apktool, adb, and OpenJDK (or JRE) for both keytool and jarsigner are installed on your system. If you are unsure how to do this, use Google. There is a decent chance OpenJDK is already installed on your system, and if you run into trouble, continue Googling. One could write an entire lengthy tutorial or blog post on dealing with Java installation, and several people already have, so go look for one of those. Also, make sure your Android can access Developer Options, and turn on the switch next to USB debugging. Again if you do not know what this is, Google is your friend.

1. Download and Disassemble

Download the version of the APK you wish to experiment on from a reliable source - **apkpure.com** is popular around here. Use the apktool tool to disassemble the APK:

```
fang:~ sn$ apktool d the_example.apk -o the_example_apk_disassembled
```

2. Modify the Disassembly

Find the proper spot to modify. Searching the smali code in the smali subdirectory for keywords such as “X509TrustManager,” “cert,” “pinning,” etc. should do it. The file we need to modify contains methods named “checkClientTrusted,” “checkServerTrusted” and “getAcceptedIssuers.” Add the “return-void” opcode before the first line of each of these three methods. The “return-void” statement is a Dalvik opcode to return ‘void’ or null, and in essence, with all three disabled, you have removed certificate pinning.

3. Reassemble the Modified APK

Reassemble the modified APK using apktool:

```
fang:~ sn$ apktool b the_example_apk_disassembled/ -o the_example_modified.
apk
```

4. Sign and Install the APK

Before the modified APK can be installed onto a device it needs to be cryptographically signed. Here are the quick steps to do this:

Generate the private key:

```
fang:~ sn$ keytool -genkey -v -keystore my-release-key.keystore -alias
alias_name -keyalg RSA -keysize 2048 -validity 10000
```

Sign the APK using the generated private key:

```
fang:~ sn$ jarsigner -verbose -sigalg SHA1withRSA -digestalg SHA1
-keystore my-release-key.keystore the_example_modified.apk
```

If you get the “Please specify alias name” use the following command instead:

```
fang:~ sn$ jarsigner -verbose -sigalg SHA1withRSA -digestalg SHA1
-keystore my-release-key.keystore the_example_modified.apk alias_name
```

The modified APK should now be signed for 10,000 days and ready to be installed onto the Android device. Attach the Android to the computer’s USB port and run:

```
fang:~ sn$ adb install the_example_modified.apk
```

After installing the modified APK, it is possible to MITM the HTTPS communications via the Wi-Fi connection - the easiest method is to use mitmproxy running as a transparent proxy.

Appendix B: Disclosure Process

Normally when reporting issues involving security, one contacts the listed address for reporting such items (usually security@, secure@, or something similar). None for Milwaukee Tool were found, however via web searching there were numerous references to **onekeyfeedback@milwaukeetool.com** as a contact for the ONE-KEY support email address. In reviews on Google Play for installing apps on the Android platform, the email address was used involving feature usage and questions about various items, so this was deemed the main contact for the app itself.

Steps were taken to ensure that during the process of reviewing and testing the application, none of the articles listed in the **Legal** document were violated.

Here is the timeline for the vulnerability disclosure process.

- 2017 Feb 2 — Initial contact with vendor via **onekeyfeedback@milwaukeetool.com** with question about app permissions and if this was the correct address to report security issues. No response.
- 2017 Feb 10 — Full report of flaws sent to **onekeyfeedback@milwaukeetool.com**. As it is late in the afternoon on a Friday, the 90 day clock for disclosure starts Monday.
- 2017 Feb 13 — 90 day countdown begins.
- 2017 Feb 20 — No response as of yet so tried using the web form on the ONE-KEY FAQ page (**milwaukeetool.com/ONEKEY/Support**) but it did not "work". Used the generic form at **milwaukeetool.com/contact** and asked if the **onekeyfeedback@milwaukeetool.com** address was the correct address for reporting security vulnerabilities.
- 2017 Feb 27 — Called 1-800-SAW-DUST for assistance since I have not received any response to contact via email or web form. Was transferred to a second level technician, where I asked for a contact for reporting security issues with their products. I spoke to a gentleman named Steve who said he would pass my contact info on to the ONEKEY department, and they would contact me.

- 2017 Mar 2 — No response, called 1-800-SAW-DUST back to try again. Waited on hold for 22 minutes, was disconnected.

- 2017 Mar 3 — An attempt at contact in a public forum. I wrote a 3 out of 5 star review on Google Play, stating that I had found some security issues, pointed out the contact above, and asked for proper contact information. I did not state whether the security issues were severe or not, just that they'd probably want to address them. No response.

- 2017 Mar 20 — Still no response. I really do not want an IoT-type vendor's initial contact with the security community to be negative, so instead of just waiting for the clock to run out, I have reported the issue to CERT via the **cert@cert.org** address. They have a usual wait time of 45 days, which lines up with where we are with our timeline.

- 2017 Mar 21 — Received a response from CERT (tracking with VU#955683). They will attempt to reach the vendor, and will keep me informed as to their progress.

- 2017 Apr 14 — CERT has made contact with the vendor via Twitter, will copy me in on future communications.

- 2017 Apr 21 — CERT received a reply that in part stated "Any details that you can share about your discovery would be much appreciated as security of our system and users' data is very important to Milwaukee." They replied with the details of the four issues, and recommended the creation of a "security@" email alias for future contacts regarding security issues.

- 2017 May 1 — Contacted CERT and asked if they had heard anything.

- 2017 May 4 — Star Wars day. CERT advised no word from the vendor, they will contact them and advise that disclosure is imminent. Will plan for next week unless the vendor replies with progress.

- 2017 May 5 — CERT advised that the vendor responded to their request for an update and has asked for more time, citing the April 14th twitter message as first contact they were aware of. A representative named Chad from Milwaukee Tool has been given my contact info and told to contact me directly. All vulnerabilities have been confirmed by the vendor, hoping for a meeting to straighten things out.

- 2017 May 13 — Ninety day mark.
- 2017 May 16 — Final attempt, contacted Chad via the email supplied from CERT, explained our intent to publish our findings.
- 2017 May 23 — No response. Will start the wheels in motion for publishing.
- 2017 May 24 — Notified CERT of intentions, asked for an update on CVE IDs. CVE IDs CVE-2017-3214 and CVE-2017-3215 were assigned to issue 1 and 2 respectively.
- 2017 Jun 6 — Notified CERT of date to publish, and URL of main blog post when it is published. Publish date is June 19, 2017.
- 2017 Jun 19 — Published report.

Mark Loveless

Senior Security Researcher
@simplenomad

Mark Loveless is a Duo Labs researcher who also goes by the name Simple Nomad on the interwebs. He is not overly paranoid in spite of the fact that evil alien robots are stealing his luggage when he travels.

Duo Labs

duo.com/labs
@duo_labs

Duo Labs is the security research team at Duo Security. Duo Labs is half the reason Duo Security has a legal team. Check out their other research and follow them on the twitter.

Our mission is to protect your mission.

Experience advanced two-factor authentication, endpoint visibility, user policies and much more with your free 30 day trial.

Try it today at **duo.com**.

Duo Security makes security painless, so you can focus on what's important. Our scalable, cloud-based **Trusted Access** platform addresses security threats before they become a problem, by verifying the identity of your users and the health of their devices before they connect to the applications you want them to access.

Thousands of organizations worldwide use Duo, including Facebook, Toyota, Panasonic and MIT. Duo is backed by Google Ventures, True Ventures, Radar Partners, Redpoint Ventures and Benchmark. We're located from coast to coast and across the sea.

Follow @duosec and @duo_labs on Twitter.

The Trusted Access Company