C L O C K   L O C K I N G   B E A T S

Exploring the Android Kernel and Processor Interactions

PROJECT REPORT & FINDINGS

A DELIVERABLE PRODUCT OF:

MONKWORKS, LLC

JOSH "M0NK" THOMAS

Table of Contents

Overview of Research

Review of the project and goals

(The following section is an excerpt from the initial proposal, included herein for context)

<u>Executive Summary</u>

This is a proposal to explore the manipulation and detection of smartphone processor clock controls to hide running processes from end users.

Smartphones have undoubtedly infiltrated our daily lives. We carry them everywhere and use them for almost every facet of our daily toils. As such, we have become infinitely familiar with how these devices behave in our hands and would certainly notice if they started acting quirky. Before the information about viruses, rootkits and malware became public we might have shrugged off degradation in the user interface or random crashes of applications to chance. Now we, as the general public, would at least consider the possibility that our beloved device was compromised once erratic behavior was noticed. As with standard computers, our paranoia leads our defensive and reactionary posture against such threats.

From the other side, developers of these nefarious tools must become creative to stay hidden and undetected in the current mobile landscape. If malware is to remain unnoticed, the device cannot suffer performance degradation in any aspect of user facing interactions. One such method to raise the sophistication bar in this cat and mouse game of process hiding is to remove any malicious processing footprint from the user space altogether. The quandary is technically how would that be possible if the software in question is to perform any processing?

The attack and defense vectors this research would like to consider begin in an entirely different problem, that of mobile battery life. Manufactures are constantly attempting to add better, faster, stronger and flashier hardware to their offerings while still attempting to keep these devices running for 8 to 14 hours a charge. The newest gadget is useless on the market if the battery only lasts an hour. One strategy that vendors utilize is to underclock the main processor on the device (typically by about 10% down from the datasheet specifications of the embedded hardware). The market is mixed with static and dynamic solutions for underclocking hardware, but the common theme discovered during research is that all vendors attempt to “dial in” the manipulations based on the device, its components and the battery life they want to achieve for marketing purposes.

Interestingly enough for real world applications, preliminary research has shown that minor changes to processor clocking speeds do not have a huge effect on overall battery drain. Returning the clock to datasheet speeds or even slightly overclocking tends to shorten the battery life by minutes and not the previously assumed hours. What actually kills battery life is the locking and binning of processor speed settings into poorly chosen ranges.

Moving back to the attacker / developer standpoint, we can now begin to see the possibilities for hiding malicious code from a user by simply adding processing cycles into the pool. An end user will not notice the consumption of processor cycles if they were never present to begin with. If the developer is smart and conservative, the user will also never see a significant change in battery life. The developer can, in a manner of speaking, pull processing cycles out of thin air and use them unnoticed.

As a paranoid security professional, this idea bothers me. I've seen no valid research into this concept, from either the offensive or defensive spaces. We cannot defend against an attack without tools, and we cannot make tools for defense until we understand the problem. And thus, the proposal you are reading.

The Clock Locking project will attempt to create a proof of concept implementation of how an attacker would manipulate processor speed changes to hide running code. This proof of concept will allow the researcher to understand exactly what it would take to deploy this type of attack in the wild. With fresh knowledge in hand, the main focus of the Clock Locking project would then be the generation of a discovery tool for such manipulations.

Technical Descriptions

In reality, large scale manufacturing of electronic components is never the hard science we expect. Production runs, yields and quality vary heavily based on an almost infinite number of factors, from weather to silicon quality. While the variance is not directly predictable, it is accounted for during production. Specifically for processor generation, each chip produced is checked for a minimum speed capability and if that threshold is met, the package is stamped and shipped as such. The actual speed capability of each chip is inconsequential as long as it meets the minimum; the package will be branded, firmware locked at the threshold speed and sold to the market as such.

From a software (and especially kernel) perspective, the processor is expected to behave at the reported speed. Typically, if a processor reports itself to be capable of 2.0ghz, the kernel will assume 2.0ghz and never attempt to run faster. Underclocking is achieved by simply telling the kernel the chip is capable of running at a maximum of 1.8ghz (or some other lesser number). In this sense, the kernel is quite dumb and just utilizes the cycles it is told exist.

In the Android project, as with most active modification and gaming communities, the concept of overclocking and voltage manipulation are fairly common and well documented online. A quick Google search will showcase many tutorials and arguments about technique and implementation, and will probably leave the researcher curious about the seeming black art of tuning an overclocked device for stable performance and speed. Regardless of approach, this concept is far from novel.

What is highly uncommon, at least outside of the hypervisor space, is constraining a process inside a resource window and not allowing the process to consume unbounded CPU cycles. A developer of high-end malware would see no benefit in regards to detection evasion if they simply overclocked the processor to run faster. To move the solution to its elegant conclusion, they must ensure:

- The processing cycles gained in the delta between stock and CPU overclocking must always be consumed by the malicious code.
- The malicious code must never utilize more cycles than the delta allows.
- The malicious code must conceal the fact that the processor is unlocked and overclocked.
- The malicious code must correctly interleave hidden processing with normal processing at the kernel level to successfully evade detection.

With this adversary in mind, a defensive tool chain would need to detect concealed processor speed manipulations and possibly flag processing threads that behave in resource locked manners. Furthermore, a defensive tool must be able to analyze the current operating environment and discover if the CPU settings have been modified without the ability to trust the actual computing environment. Lastly, the defensive toolset must not interfere with the typical CPU speed bursting capabilities of modern hardware; it must only report variances that appear to be malicious in nature.

Technical Objectives

The Clock Locking project will document a deep-dive exploration into smartphone processor clock, speed and voltage manipulation techniques. The tools and documentation will catalogue various means to subvert detection of running processes on mobile devices and how to uncover them.

The project will also generate a framework for applying processing power resource windows to mobile devices. While this concept is somewhat understood in the hypervisor and virtual machine space, it is very uncommon in the smartphone runtime environment. Beyond simply providing bounds for processing thread execution, this framework must correctly interleave running processes in a time-shared, scheduled manner. This framework will have uses beyond this specific project, especially in the realms of advanced mobile AV techniques.

Finally, the Clock Locking project will produce a useful defensive tool for detecting advanced malware and rootkits on mobile devices. This tool will have the ability to harness the same advanced CPU manipulation techniques to protect an end user device as malicious code does to compromise it. The tool will harness the developed frameworks to provide an unobtrusive background process to protect the smartphone during runtime.

Selection of Device

The Sony Xperia Z

The main device utilized in the research is the Sony Xperia Z.

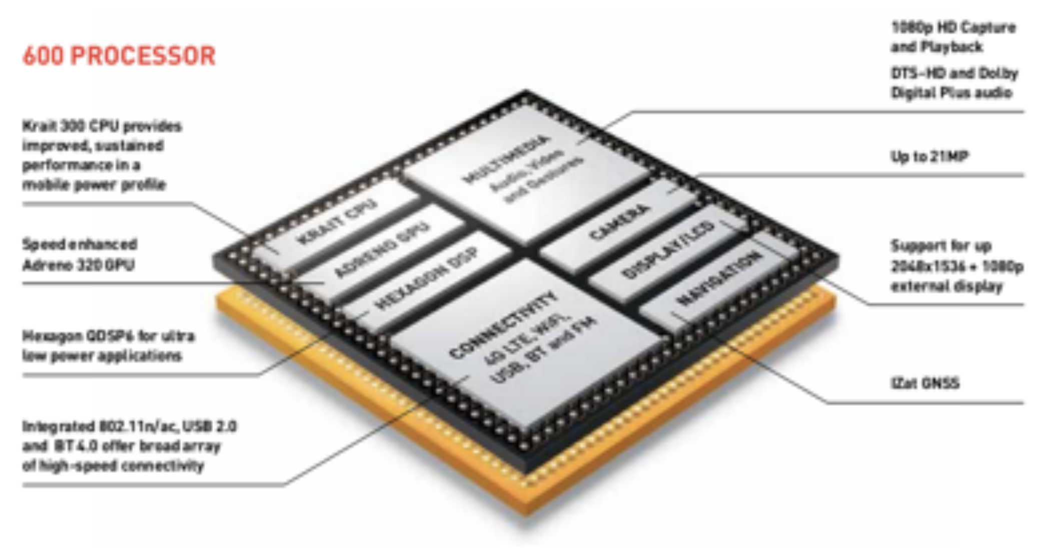
Sony Xperia Z Internals

The Z was released by Sony as part of the Xperia line in February 2013. Out of the box it ran Android 4.1 and has since been upgraded to Android 4.1.2. The device is powered by a non removable 2400 mAh battery, contains a 1.5 GHz quad-core Krait processor internal to the Qualcomm Snapdragon S4 pro chip. As expected, the device contains the full spectrum of internal components common to high end smartphones. Of note, the Xperia Z is mostly waterproof and as such is very difficult to open and is not designed to allow for any internal part replacements.

AOKP Support for the Xperia Z

The AOKP project officially supports this device. The generic code base (AOKP is based on Cyanogenmod, which in turn is based on pure AOSP) does not offer *Project Burner* anything special code or API wise. What is gained from using this open ROM is a cross platform source code and compilation environment that has been standardized and cleaned of vendor specific files. This allows the research as a whole to focus on pure Android kernel modifications without the distractions of vendor specific modifications. Given the nature of this project, it is currently expected that any PoC that functions on a device running an AOKP Rom variant will function on the native / stock device with the same capabilities. This is ensured because the AOKP changes overall are irrelevant to the specific kernel code this project is concerned with.

The Qualcomm Snapdragon S4 Pro SoC


600 PROCESSOR
Krait 300 CPU provides improved, sustained performance in a mobile power profile
Speed enhanced Adreno 320 GPU
Hexagon QDSP6 for ultra low power applications
Integrated 802.11n/ac, USB 2.0 and BT 4.0 offer broad array of high-speed connectivity
1080p HD Capture and Playback
DTS-HD and Dolby Digital Plus audio
Up to 21MP
Support for up 2048x1536 + 1080p external display
IZat GNSS
KRAIT CPU
ADRENO GPU
HEXAGON DSP
MULTIMEDIA Audio, Video and Gestures
CAMERA
DISPLAY/LCD
NAVIGATION
CONNECTIVITY 4G LTE, WiFi, USB, BT and FM

Qualcomm sample SoC diagram

The Snapdragon S4 Pro is powering the Sony Xperia Z as well as the Xperia ZL, The HTC Droid DNA, The Google Nexus 4 and the LG Optimus G. The detailed data sheets on this SoC are non-existent from Qualcomm but the promotional sales information details that the 28mm SoC contains:

- Up to 1.7 Ghz dual or quad core Krait CPU
- Adreno 320 GPU
- 3G/4G & LTE support
- USB 2.0 High Speed OTG support
- WiFi / Bluetooth / GPS / high end camera support

[http://www.qualcomm.com/snapdragon/processors/s4/specs]

In general, the SoC contains the bulk of the processing capabilities for the phone. The specific S4 pro chipset running in the Xperia Z is the APQ8064.

Concepts of the Project

Original Ideas

The original idea behind *Clock Locking Beats* was to inject a programatic shim into the Android kernel that allowed for arbitrary overclocking. The artificially generated processing capability increase was then slated to be thread locked to a specific task (either the offensive or defensive PoC) inside the kernel. This capability would allow for either a nefarious implant or a defensive tool to run outside of the typical user land space in a hidden and unnoticeable manner. The speed increase gained would be used solely for the implanted code and no processing cycles normally available to the device or the user would be affected. The original proposal included the idea of analyzing the overclocking mechanism to ensure that little battery drain or usage occurred due specifically to the modifications.

Evolution of the Project

During the research portion of this project, a handful of alternate methods were also explored. While the main focus was to perform this research bound inside the Android kernel, analysis was also performed on the bootchain of the device. This analysis proved bountiful for future ideas and research, but did not yield a direct product during this engagement. While it is entirely possible to inject a hypervizor into the Secure Boot 3.0 bootchain used by Qualcomm powered Android devices, the application vector requires a reliable exploit / vulnerability and the removal of the “root of trust” powered bootchain.

This methodology would behave as such:

- The Android device begins to boot (physical power button pressed from a cold / off state)
- Somewhere in the bootchain (prior to the 3rd secondary bootloader), the CLB implant would need to pause the boot process and inject a small footprint hypervizor
- The hypervizor would load the CLB thread to run external to the kernel
- The CLB thread would overclock the processor for the first time
- The hypervizor would then continue the normal boot process and allow Android to load itself
- The CLB thread would simply Man in the Middle all voltage, clock cycle and CPU speed requests from the Android kernel and return modified values.

While this concept was outside of the original CLB scope, the path was explored during the research phase to ensure that no avenue existed to this functionality that did not require bootchain exploitation.

Future Explorations

This project was intended to generate functional Proof of Concept implementations for both offensive and defensive capabilities of the CLB mindset, and it is believed that goal was accomplished. MonkWorks intends to continue this research path and eventually release a tool that injects CLB into the Qualcomm Secure Boot 3.0 bootchain. The researcher also intends to continue the analysis and metric / algorithm creation process for the defensive CPU attestation portion of this project.

The researcher will also continue to develop an overclocking shim for the kernel that will allow for hidden overclocking by MITM the regulator / governor calls and replacing the requests with modified information.

The Overclocking Mechanism

Note To Reader: The overclocking mechanism provided is divergent from most overclocking mechanisms available on the open market. For the remainder of this project, either overclocking approach will suffice with the same basic results.

The overclocking mechanism delivered during the runtime of this project simply showcases how overclocking works on Android. Simply put, the kernel regulates the voltages supplied to the processor and adjusts the clock settings to match various requested states of speed. The governor framework then analyzes the runtime parameters of the kernel and adjusts the speed of the processor cores accordingly. To overclock the processor cores, the software is simply required to inject another voltage / speed pairing into the system.

For the proof of concept code delivered, CLB took a slightly different approach to overclocking. The modified kernel source adjusts every possible speed and voltage setting and increases them by a static 10%. This methodology is not suggested for regular use, but does showcase the ability to gain significant speed from the stock processor without a large hit to the battery.

As discussed in the “Future Explorations” section above, MonkWorks researchers also designed a conceptual solution for processor overclocking using a light weight shim. This shim (in the form of a side loadable kernel module), will allow for dynamic percentage based overclocking. The unique component of the shim lies in the functionality of not allowing the rest of the kernel and regulation system to realize the device is overclocked. This is accomplished by intercepting all messages sent to and from the regulator and governor sub-systems and replacing the contents of those messages at runtime.

More detailed information about the overclocking used in this project can be found in the *source_and_notes_overclocking* source code directory attached to this report.

Understanding the Offensive PoC

Note To Reader: Full build and runtime instructions for the tools were provided during initial delivery. These instructions will also be attached to the delivered source along with this report. This section will explain "what" the code does and "how" it does it without necessarily explaining "how" to run the code itself.

The offensive proof of concept tool is a collection of 3 discrete programs that are meant to be run in sequence. The PoC is designed to showcase how to programmatically lock a computationally complex thread of execution to a desired processor utilization. This action is performed thus:

- The *oc-me* program over clocks the Qualcomm Snapdragon processor by a fixed percentage as supplied during execution time (README files suggest 5-10% overclocking)
- The *collatz-kit* program randomly loops through Collatz sequences, helplessly searching for a deviation from the algorithm. While Collatz sequences are based upon simplistic mathematical principles, the algorithm has been optimized to consume as many CPU cycles as possible (if left unfettered and unbounded).
- The *cpulimit-simplified* program injects itself into the kernel and modifies the interleave mechanisms provided to ensure the targeted program (*collatz-kit*) only utilizes a user specified processor percentage (assumed to match the value provided to *oc-me*).

The *collatz-kit* program will consume an average of greater than 50% of all available CPU cores during runtime if left unchecked and unrestrained. When the entire collection of tools is run in sequence, the program can trivially be constrained to the defined 5% or 10% CPU usage. Given the overclocking mechanism injected, this collection can arguably run undetected from the end user with no artifacts or lags alluding the the intense processing that is occurring on the device.

While the offensive PoC was specifically designed to be of little use in the real world, the software does prove the solid conceptual foundations of the proposed solution.

More detailed information about the offensive PoC portion of this project can be found in the *source_and_notes_offensive_poc* source code directory attached to this report.

Understanding the Defensive PoC

Note To Reader: Full build and runtime instructions for the tools were provided during initial delivery. These instructions will also be attached to the delivered source along with this report. This section will explain "what" the code does and "how" it does it without necessarily explaining "how" to run the code itself.

The defensive proof of concept tool is a solitary kernel module that preforms a very simplistic but highly targeted form of attestation on the main processing cores. The *attestify* kernel module performs a series of three discrete tests thousands of times in a loop and records the temporal bounds of each test . The module is designed to generate a baseline evaluation of processor capabilities and speeds. This theory of attestation can illuminate specific types of temporal drifts away from the standard averages, and that is how *attestify* can determine if a CPU core in potentially running hidden code.

The *attestify* module is comprised of 3 discrete "tests" that are run in loops. The "tests" themselves are mathematical calculations that were selected based on the approximate number of CPU cycles they require. The tests are defined as:

- Short test: This test is designed to take less than 10 CPU cycles. The short length of this test normally ensures each loop is completed without a single context shift in the processor.
- Medium test: This test is designed to take more than 10 CPU cycles but less than 100 cycles. The medium length of this test normally ensures each loop captures at least one CPU context shift.
- Long test: This test is designed to take significantly more than 100 CPU cycles. The long length of this test normally ensures each loop contains multiple process and context shifts.

The *attestify* module records the time (in nanoseconds) that each discrete test required for completion. As each test is run inside multiple nested loops thousands of times, the module builds a runtime fingerprint of temporal data and metrics associated with a given processor and device. This data is then analyzed and a simple metric collection (including averages and standard deviations) is presented to the user. The usefulness of this tool is the ability to see a processor core shift from behaving a certain way when normal software is run to an unexpected way when other processes deviate from the standard practices used across Android and linux development. By relying upon baseline metrics calculated against a large running sample size, the functionality and usability of the *Attestify* module is unaffected by CPU utilization rates.

More detailed information about the defensive PoC portion of this project can be found in the *source_and_notes_defensive_poc* source code directory attached to this report.