The Advanced Persistent Threat You Have: Google Chrome

Todd Heberlein
LTH@NetSQ.com

The Advanced Persistent Threat (APT) has become the watchword for cyber espionage laying waste to our national and economic security. Do you have APTs inside your organization right now? How can you be confident of your answer? I argue that you probably already have a "benign APT" inside your organization, and your ability to detect, analyze, and understand this benign APT's actions will tell you whether you have a chance to do the same for malicious APTs. That benign APT is Google's software update system. I pose key questions that your organization should be able to answer about this activity. I present a summary of my findings and a somewhat detailed analysis of Google's update activity. To determine if your organization is prepared for a modern threat, you should consider a similar exercise with the data you currently collect and the tools you use to analyze that data. If you fail with the Google APT, you will probably fail with a real APT.

1 The Threat

The Advanced Persistent Threat (APT) has become the watchword for today's cyber espionage. It frequently involves a piece of malware or group of malware programs that can evade detection (e.g., by antivirus software), remain in target systems for long periods of time (e.g., months or years), and reach out across the Internet to exfiltrate data (e.g., Word documents or emails) or allow an attacker to exert further control of the system.

Economic losses from attacks involving APTs vary wildly, from just a few billion to hundreds of billions of dollars per year. But with terabytes of military and industrial secrets being siphoned out of the country every year, APTs clearly represent strategic military and economic threats.

APT malware often operate for months and years within a site before being discovered. Frequently, perhaps over 90% of the time, the organizations don't detect the threat but rather they are notified by third parties such as the FBI that they have been compromised. And these are not organizations who do not care about security; they typically have fully up to date antivirus software, firewalls, network IPSes with the latest signatures, and regularly patch their software.

Why are organizations unable to reliably, and in a timely fashion, detect, analyze, and understand APT activities occurring in their systems? While there are numerous possible answers, I believe one critical answer is that they are not looking at the right data.

It is like the old joke of a drunk looking for his keys under a streetlamp. A passerby stops to help him search, and after awhile the passerby asks, "Are you sure you lost your keys here?" The drunk answers, "No, I lost them over there, but the light is better here." Organizations are

looking where the light is better (e.g., network traffic at the perimeter) and not where the keys actually are.

Is your organization able to detect and analyze APT activity inside your network? If you were hit, would you know? How can you be confident of your answers?

2 The Model APT

Chances are you probably already have a benign APT in your organization. We know it is benign because the motto of the organization who has inserted this APT into your network is "Don't be evil." Yes, I refer to Google, and in particular, Google Chrome's update system.

Why do I refer to Google Chrome's update system as an APT? Because in many ways it behaves just like an APT. So much so in fact, I consider it a model APT for both attackers and defenders. If you are a bad guy and want to get into the APT game, analyzing the Google software update system would be a good place to start. If you are a good guy and want to defend your network, you can test your mettle by trying to detect and analyze Google changing software on your systems.

Here are just some of the reasons Google's software update system can serve as a model APT. (1) There is a bootstrapping program, GoogleSoftwareUpdateAgent, buried deeply in a user's home directory. Few people know it is there, even fewer probably know when it becomes active. It neither requests permission to modify your system nor notifies you when it does. (2) Detailed analysis of this bootstrapping program cannot reveal what it may do because actual actions (via programs & scripts in this case) are only received from the Internet at runtime. (3) A critical executable (one you trust with your passwords for many important network services) will be modified.

Google's software system is also useful as a model APT because your organization probably has it running on production systems. The same systems that will be targeted by malicious APTs. Practicing protection, detection, and analysis on isolated testbeds often lead to skewed conclusions and expectations because it is the noise, complexity, and chaos of production systems that often make security so challenging. Furthermore, Google's software update system, especially for Chrome, modifies your systems frequently, so you don't have to wait for a relatively rare actual attack to test and practice your skills.

3 The Challenge

My claim is, as a defender, you should be able to answer two questions: "So what?" and "How did this happen?"

So what? This is a "flow forward" question. First, your security architecture should be able to detect the execution of the bootstrapping program. Second, and more importantly, once identified, you should be able to answer the question "So what?" What are the implications of

this program running? Does it read Word documents and exfiltrated them out of your system? Does it modify (Trojan?) your system? You cannot determine if the bootstrapping program is part of an attack or not unless you can understand the downstream implications. In other words, you need to be able to concretely answer the question "So what?"

How did this happen? This is a "flow backwards" question. First, your security architecture should be able to detect the replacement of key resources like an important executable. Second, once the change is detected, you should be able to answer the question "How did this happen?" What was the sequence of processes that led to replacement? Where did the file come from that replaces the resource or executable? You cannot prevent re-infection (or re-Trojaning) unless you can answer the question "How did this happen?"

If you cannot answer these questions for Google's software update, you are probably not able to successfully detect, analyze, and respond to malicious APTs.

(Note: If you can perform the detailed analysis and answer the questions "So what?" and "How did this happen?", are you prepared for all threats? No. Being able to analyze Google's activity is a necessary but not sufficient capability for facing modern cyber attacks. I only claim if you can't do a similar level of analysis on this benign APT, you are not prepared for malicious APTs.)

4 Summary of Observations

I did my own analysis of Google's software update using Apple's BSM audit trail as a data source and my own audit analysis software. A more detailed description follows in the next section. In this section I describe some of my key observations based my analysis. References to various steps in the "attack" are for labeled activity in Figure 1.

The actual changes to the system is made not by the malware (or Google supplied code) but by legitimate system applications. For example, the copy program, cp, changes waitCursor.png (step 24), and the remote sync program, rsync, modifies the "Google Chrome" executable (step 36). If your security software detects the change to the system (e.g., replacement of the Chrome executable), and then analyzes the program making the change, the security software would detect no malicious signatures in the application because there are none. The rsync program is a perfectly legitimate, non-malicious application. No security software (or human analyst), by simply analyzing the execution of either of these programs (cp and rsync), can determine whether these actions are part of an attack or not. These are perfectly legitimate programs being used as they were intended to be used.

The original source of the data that replaced the critical executable cannot be determined by simply looking at the final change (step 36). For example, while security software that intercepts the attempt to change the application binary "Google Chrome" could determine that that program is being replaced with the file ".Google Chrome.OQwVWD", the original source of that replacement binary is separated from the final action by five intermediate files (steps 10, 12, 17, 29, 35), one network connection (step 8), and the execution of six programs (steps 6, 2, 15,

26, 30, 33), two of which never existed on the system before (steps 6 and 26). No security software (or human analyst), by observing the final step replacing the executable, can determine the original source of the new executable code.

Detailed analysis of the bootstrapping program (in this case GoogleSoftwareUpdateAgent) cannot reveal what will happen when it is run. Most of the activity changing the system is directed by programs and scripts that do not exist on the system prior to execution of GoogleSoftwareUpdateAgent. For example, the ksurl, .keystone_install, dispatcher.sh, and xzdec programs that download the files, transform the files, and ultimately replaced the "Google Chrome" executable with one of the files are all downloaded on the fly. Thus, examination of GoogleSoftwareUpdateAgent prior to its execution or after the system's modification cannot reveal its role in any system modifications. No security software (or human analyst), by simply analyzing the program GoogleSoftwareUpdateAgent, can determine how its execution will affect the system.

To avoid false negative and false positives, detection software must understand the deep history of control and data leading to modifications of the system. To avoid false negatives (i.e., not reporting an actual attack), the detection software must initially identify the modification of a binary as a potential attack. Routinely ignoring changes to binaries on the system, especially one as critical as a web browser used to access many sensitive online services, will lead to disaster. However, most of these changes (hopefully) are legitimate software updates, so reporting these changes would be false positives (reporting legitimate actions as an attack). Only by analyzing the history leading to the final step can the detection software have a chance of properly determining if the change to the system is a legitimate software update or an attack.

The control and data flow graph leading to changes to the system also identifies resources that must be protected from undetected modification. For example, if your detection software detects changes to the system but dismisses those changes that can trace their history back to GoogleSoftwareUpdateAgent, then your detection software implicitly trusts GoogleSoftwareUpdateAgent. Any malicious modification to this specific application can lead to any future changes to your system (e.g., Trojaning of your web browser) being undetectable (or at least, unreported). Unfortunately, GoogleSoftwareUpdateAgent, lying nine folders deep inside my home directory, is writeable by any software running as me, so any program I run that has a vulnerability can be exploited by an attacker to modify this critical system-changing program. Thus, if the security system suppresses actions that can trace their control back to this bootstrapping program (which is now controlled by an attacker), all subsequent malicious actions will go unreported. All elements in the control and data flow graphs that can lead to changes in the system must be protected, or at least monitored, with the same vigilance as any system critical program or file.

5 Chrome Update System Analysis

In this section I walk through the complex graph shown in Figure 1. The graph is divided into three layers: the network layer at the top, the process layer in the middle, and the file system layer at the bottom. The purpose of this graph is to show how one resource file, waitCursor.png, and one executable, "Google Chrome", are replaced on your system through a complex series of steps that occur without any user knowing about it. The actual graph is actually much more complex. I only selected the relevant processes and files needed to understand how two files were changed during the update process. In actuality, many more files are modified.

The data for this graph came only from Apple's BSM audit data. No other data sources were used (e.g., source or binary code analysis, disk forensics, etc.). I used our Audit Explorer tool to analyze the audit data.

I begin with the operating system's master launchd process on the far left of the process tree.

Steps	Explanations
(1)	The root process is labeled "unknown" because it started before the audit system started (thus the audit data does not include the name of the program execution), but from experience I know this is the top launchd service responsible for starting most other processes that will run on this computer.
(2)	When I log in, another launchd process is run; this one is responsible for launching processes for me.
(3)	The second launchd automatically, and behind the scenes, starts the GoogleSoftwareUpdateAgent (GSUA). The user never sees any evidence of this. No window appears. No approval is asked for. No notification is given. This is the bootstrapping program that starts the entire cascade of new processes and changes to the system.
(4)	GSUA first contacts a remote server over an encrypted connection. This would appear as a normal outbound secure web connection. Most of the time the process exits after this connection. No new processes are created; no changes to the system are made. But sometimes, as in this case, a cascade of new processes and system changes follow.
(5)	The cascade of changes begins with GSUA creating a new program, ksurl, written in a temporary directory.
(6-7)	GSUA creates a new process (6) that executes (7) the ksurl program created in step 5.
(8-10)	ksurl connects to a two servers over normal web connections (8,9), downloading in the process the file com.google.Chrome.dmg, saving it in yet another temporary directory (10).
(11-12)	GSUA copies the newly downloaded disk image file in the temporary directory (11) to a directory inside my home directory (12).

(13-17) GSUA creates several instances of the hdiutil program to examine the disk image file created in step 12. The third instance (13) begins a cascade of diskimages-helper processes (14,15) that eventually reads the disk image file in my home directory (16) and attaches a filesystem named UpdateEngine-mount.35TV2rg29j under the /tmp directory (17). All directories and files to the right of this mount point are created at this time.

(18-19) GSUA creates a new process (18) that executes the program .keystone_install (19), a program that did not exist on the system until its filesystem tree was attached in step 17. .keystone_install creates a large number of helper programs that make various changes to the system; just a small portion of those helper programs are shown here.

(20-21) .keystone_install creates a new process (20) that executes the shell script dirpatcher.sh (21), a program that did not exist on the system prior to step 17.

(22-24) dirpatcher.sh calls the copy program (22), which copies the file "waitCursor.png $raw" (23) from the attached file system to waitCursor.png (24) inside the "Google Chrome" application bundle in the /Applications directory. This is a change to a resource of a critical application, using a non-malware program (cp), by copying data that can (after many steps) trace its origin back to the connections made by ksurl (8,9).

(25) .keystone_install creates another instance of dirpatcher.sh.

(26-29) dirpatcher.sh creates a new process (26) that executes the program xzdec (27), another program that did not exists prior to step 17. xzdec decompresses the file "Google Chrome$xz" (28), writing the results to "Google Chrome" (29) under a fourth temporary directory structure under /tmp.

(30-36) .keystone_install creates an instances of rsync (30), which in turn spawns two additional instances of rsync (32,33). The result is that the file "Google Chrome" created in step 29 replaces the existing "Google Chrome" executable in the /Applications directory (36). An interesting twist is that the source "Google Chrome" file is opened for reading by the first rsync process (31) and written by the third rsync process (35). Presumably the file descriptor for the source file is passed to the third rsync process (34).

6 Slices

Any user looking at a graph as complex as Figure 1 will probably give up, and, as previously stated, this is actually a greatly simplified representation. The key, however, isn't that this graph represents a graphical interface for a user but that this graph represents the underlying data structures that must exist in order to answer the questions "So what?" and "How did this happen?" Looking at slices through this graph can accelerate understanding of the complex situations. Figures 2 and 3 represent two potential slices through the graph, both are designed to answer the question "How did this happen?"

Figure 2 represents a data flow slice. In this case, you are notified that the “Google Chrome” executable has been modified (step 36 on the right), and you want to know where the new program came from. By doing a backwards data flow analysis I can trace each file and its transformation (e.g., download, copy, unpack, decompress, etc.) to its original source – the network connection (step 8).

Figure 3 represents a control flow slice. Again, we begin with the final step, rsync replacing the “Google Chrome” executable, and follow the trail backwards. Instead of following the data however, process creation steps are followed. This flow shows the order of processes that eventually gave rise to the final rsync process that made the modification.

Each subgraph provides a different view of what happened. Neither provides a complete explanation though. In practice we have found that control flow analysis is much easier to automate and already exists in the Audit Explorer program.

7 Conclusions

The analysis of a program is insufficient to determine if that program is malicious or not. The inability of today’s antivirus industry to effectively detect modern malware certainly demonstrates this fact, but more importantly is the fact that malware does nothing unique that legitimate programs don’t do. Malware reads and writes files, starts and accepts network connections, creates new processes to carry out additional tasks, and schedules tasks to be carried out at prescribed times. These are all activities that legitimate programs do all the time as well.

The analysis of network traffic is insufficient to effectively detect malware on the system. While overtly noisy activity such as spambots can raise alarms, the more subtle activity of modern APTs can easily remain under the radar because it is too easy to blend in with normal traffic. Far too many programs establish network connections today to use that as a reliable indicator. Legitimate programs call home to see if software updates are available, connect to social networking sites to post or cull information, display help documentation stored on the web, and so on. In the case of GoogleSoftwareUpdateAgent, it makes secure/encrypted web connections infrequently, staying under the radar of volume-based network detection rules. Even if an organization uses an SSL proxy to intercept and decrypt the data over the SSL/TSL connection, an adversary can simply encrypt the data prior to transporting it over an encrypted connection.

The control and data flow graph is critical to detecting and understanding attacks on your system. A detailed examination of any single data file, program, or process is insufficient to distinguish between malicious or benign activity. It is the totality of behavior, identified by the graph, that is needed.

One of the best sources of data for building control and data flow graphs, and thus being able to detect and understand malicious activity, is the system’s audit trail data. Modern operating systems’ audit subsystems, when configured appropriately, record process creation, file

accesses, network activity, and other relevant information for building these graphs. Audit data is probably the best source of information for analyzing modern threats.

The Google software update process provides an ideal test for your organization to determine if you are prepared for APTs. It exhibits many of the traits of APT malware, and it occurs frequently enough in your production network that you should have many opportunities to test yourself. No analysis of any single piece of the activity – from the starting of the bootstrapping process to the change of the system – is sufficient to determine if this is part of an APT. You need to be able to answer the questions “So what?” and “How did this happen?” in order to be prepared for APTs.

8 Figures

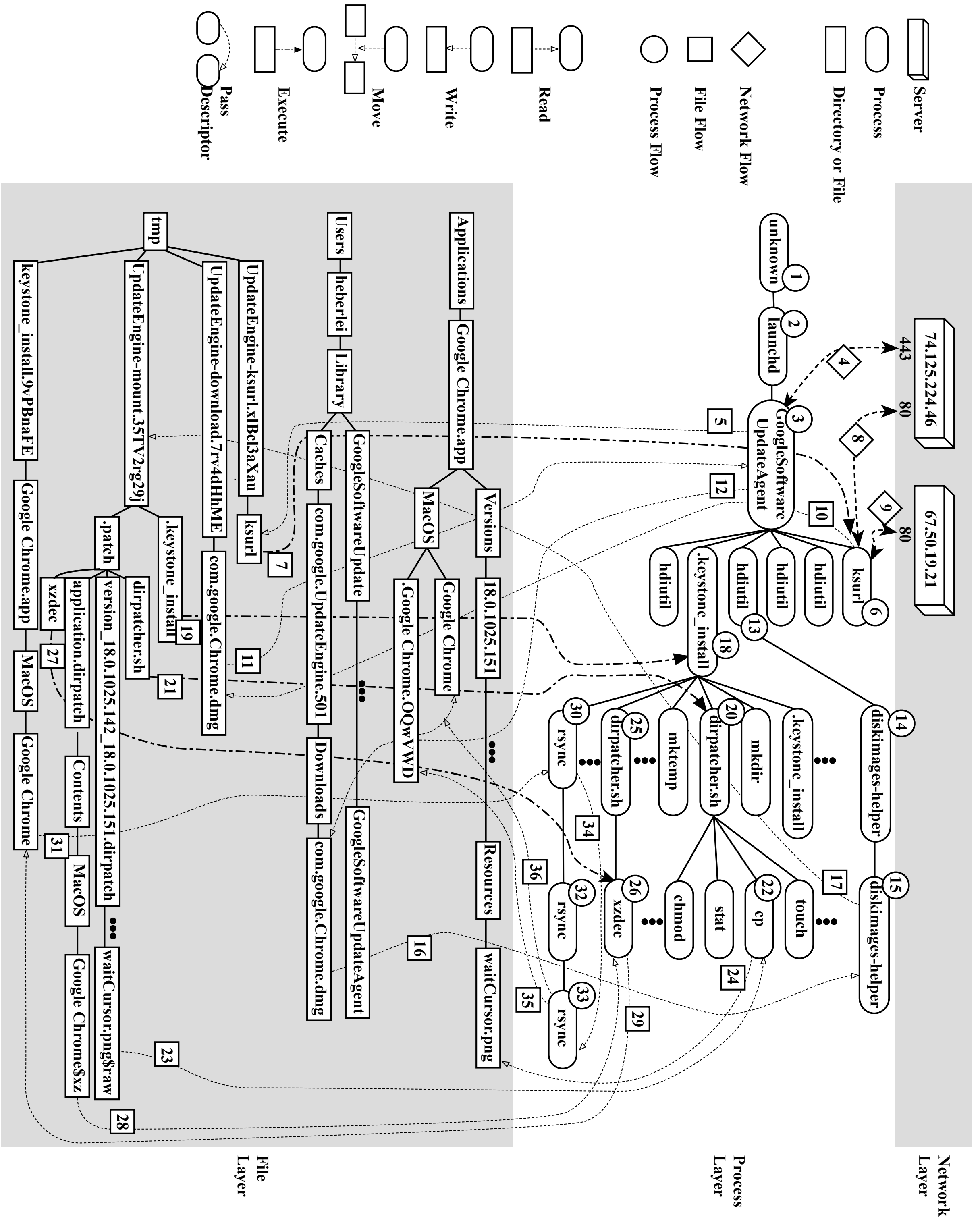

Figure 1

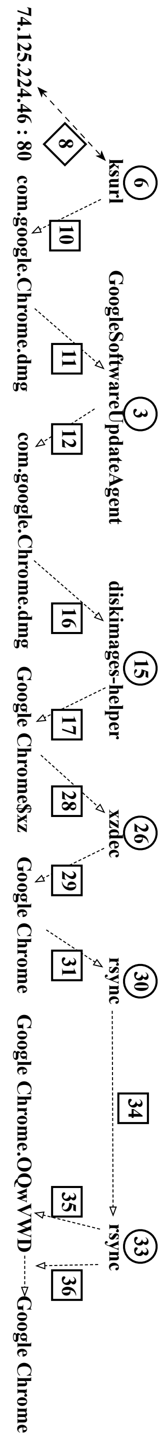

Figure 2: Data Flow

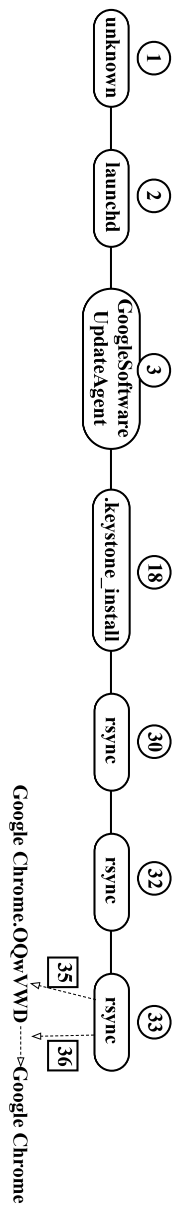

Figure 3: Control Flow

Figures 2 & 3