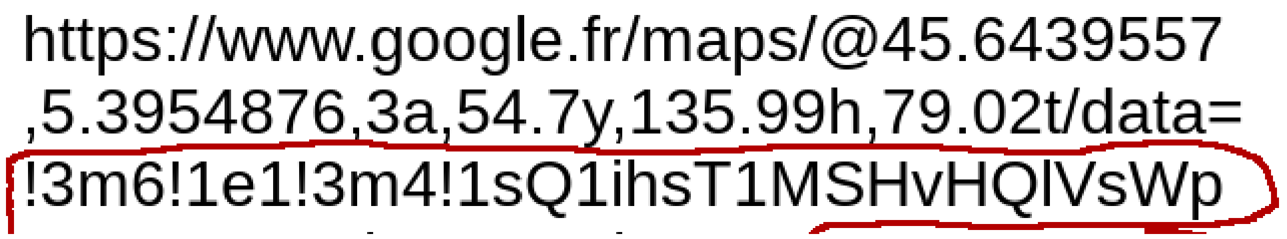
Marin Moulinier Follow
https://github.com/marin-m
Mar 9, 2017 · 16 min read

How I found a $5,000 Google Maps XSS (by fiddling with Protobuf)

A few months ago, I used Google Maps. Or maybe Google Street View, I love Street View, it's like a retrofuturistic way to teleport. Routinely, I looked at the address bar. Since sometime in 2014, parameters are not the mere query string they used to be. Instead, it's a weird mash of alphanumeric characters separated by exclamation points.

It's abstruse, it has no public documentation whatsoever, it's used by people everyday, it's a reverse engineerable protocol. That's the kind of code I want to break.



```
https://www.google.fr/maps/@45.6439557
,5.3954876,3a,54.7y,135.99h,79.02t/data=
!3m6!1e1!3m4!1sQ1ihsT1MSHvHQlVsWp
```

JKqQ!2e0!7i13312!8i6656
?
?

The Google Maps URL parameter format

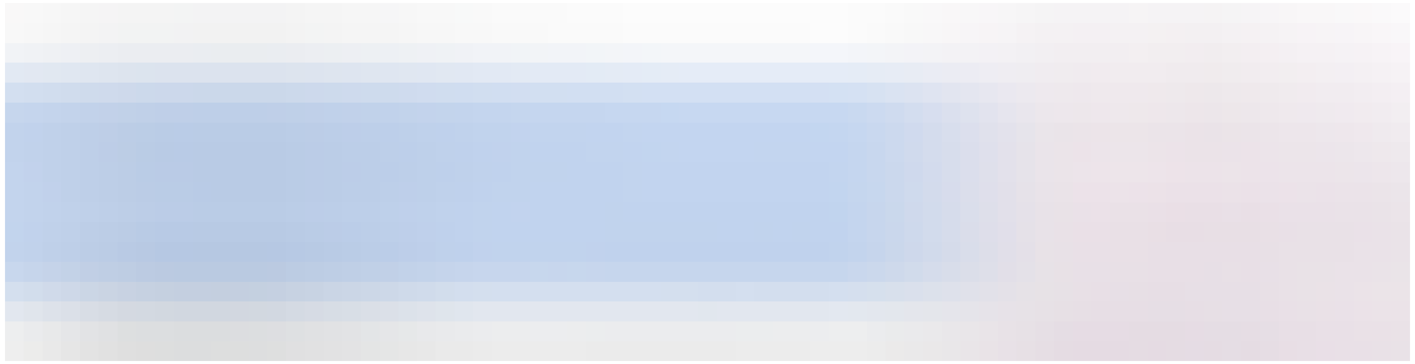
I also looked at the browser's web console. Not only requests to the AJAX API were encoded the same way, but if some of responses were images, others were a novel signature followed with cryptic binary data. With that much mystery to solve, what could restrain any more my motivation? Nothing.

Opening the hood

Google Maps is a piece of about 3.2 MB of minified Javascript. Google Earth felt heavy—yesterday's Google Maps was a dumb slippy map, today's Google Maps has a Earth mode. It renders 3D graphics with WebGL, has large tiles, requests and display caching, scheduling and planning modules implemented in Javascript, and all that stuff.

With a codebase of this size, you got to be organized. That's fortunate, all (both) major browsers have landed their own debuggers these last years, as in, breakpoints can be set in any functions you wish to, and even when an AJAX request or a DOM event fires. Most things in reverse engineering have either a static or dynamic way to solve, I'm mostly a static guy that will use dynamic analysis in extreme cases, because I like detail and the sweet feeling of losing itself to assembly. This will however get useful a bit later.

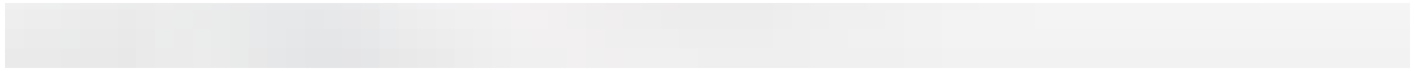
Let's get to the code—Google Maps loads about eight Javascript code files, some in workers and through XMLHttpRequest, that have linebreaks only to separate modules. Javascript modules are designated with two or three-letter codes in the URLs. It's minified with Closure Compiler, Google's minifier which can inline functions, put half of your function's statements in a conditional expression, separated with simple commas, and even more magic. Getting back code to readable enough (indented) is doable within your browser web's console, or with a simple tool like http://jsbeautifier.org/.



The requests loading Google Maps's source, and some minified code

Getting used to read this kind of output may frighten some, but is not very hard. Just like when you read assembly, you learn to go straight to the point: search for strings and constants a lot, jump from a referred to function to another, take notes in your txt, sometimes, guess what the function could be doing upon a look, and all these things.

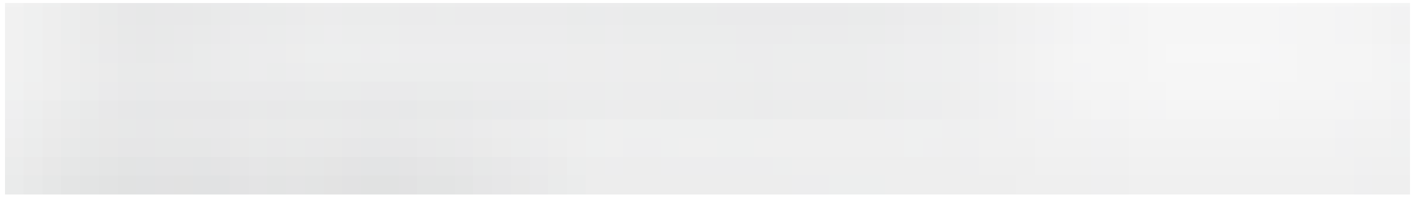
For example, what could searching for an exclamation point string do… Oh already.



Some of the code of the function that does the mysterious kind of URL encoding—apparently.

With observation, you quickly notice that URLs parameter strings (shown again in screenshot below) are made of, repeated indefinitely:

- An exclamation point: the separator.
- A number: it looks like an integer key, in a key/value pair.
- A single letter: it's a type that determines what follows. "s" is a string, "i" is an integer, but there's more…
- Either a number, or a string: the value in the key/value pair.



A screenshot from Chrome's web console, showing requests to various AJAX endpoints.

Could it be a novelty serialization format? In fact, after sorting out what I could remember about serialization, it wasn't novelty. It was Protobuf.

Protobuf is a format that's used for (storing, enabling communication with) absolutely anything at Google. It was modestly developed as version 1 in 2001, made public as version 2 in order to overthrow XML in 2008, improved somewhat as version 3 in 2016, and now it's mostly what will power your communication with "Google's new API platform" and it's used more and more in the web frontend parts of Google products, now that it is everywhere in Android's.

However, Protobuf is mostly transported (on the network) as binary data. We found text.

When you create a program using Protobuf, you first define (in a format, called .proto, that has a slightly C-looking syntax) pairs of key-value fields, where each field has a type, a name and a number. Moreover, you put these fields in blocks having names, called messages; a field can be an integer, a string, or made up of another message (it's called a nested message). There are also other types I won't describe now.

Samples of what Protobuf message definitions looks like (.proto format).

Then, you compile your .proto to code. That's right, code in any language you want (there are many bindings) that will allow you to either read or write the Protobuf messages you defined, fast.

In our example, the "Person" Protobuf message is turned into a "Person" Java class

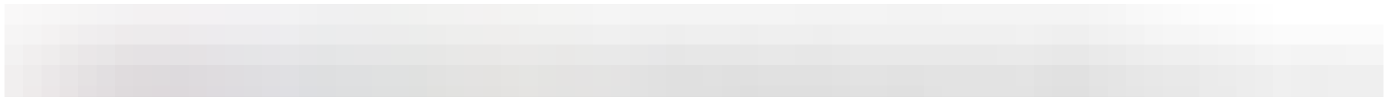
The programmer then sets Protobuf fields to its data (through calling code that was generated at the previous step), and when it's done tells the library to serialize it in a binary form.

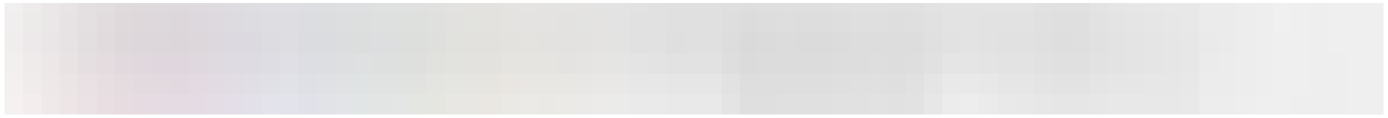
The programmer invokes the "Person" class and sets data

Resulting data is encoded as such: 8-bit tags where you have 5 bits for the field number, 3 bits for the field data type; then, the actual data: an integer or string or bytes or message or… (there are twelve different integer types, with half as much ways to encode them on the wire).

Data is serialized in a binary form

So, could the URL parameter data be the same thing, encoded using text instead?




That would have barely identical meaning, in the format used to encode URL information.

It turned out that while there are only five different data types in the binary format (0: variable-length integer, 1: fixed 64 bits, 2: string or bytes or message, 5: fixed 32 bits, 3 and 4: groups that were basically a deprecated way to define messages—just enough information to skip content you don't want or know how to read without interpreting it), there are 18 different characters that can sit as the letter designating type in Google Maps' URL.

```
// This function, found in Google Maps' code, says which type
character should map to what default field value:

dba = function(a) {
    switch (a) {
        case "d":
        case "f":
        case "i":
        case "j":
        case "u":
        case "v":
        case "x":
        case "y":
        case "g":
        case "h":
        case "n":
        case "o":
        case "e":
            return 0; // Those are integers (and enum)...
        case "s":
        case "z":
        case "B":
            return ""; // Those are string/bytes...
        case "b":
            return !1; // This one is boolean...
        default:
            return null // And there's also "m" for messages, 
referenced in further serialization code
    }
};
```

That’s just about the number of types you can use when you define fields in your .proto. But, which one maps to which? Also, could we reconstruct all the fields used in Google Maps’ URLs into readable .proto definitions; so we can **edit and replay** messages, and find possibly hidden fields? Which approach to use?

We'll reply to this very soon. No, we'll think about this right now: the function calls that defines Protobuf messages and their fields seem to be scattered around multiple functions, across Javascript code. It would be hard to scrape with a bunch of regexes, and we are lazy today.

We don't want to make sense of this.

What about a dynamic approach: positioning ahead of the function that serializes the “!”-separated URL data, waiting for structure information to come, somehow automating the use of the debugger? Communicating with that magic silicon-traversing glue called code, that allows you to talk to machines like you do with humans?

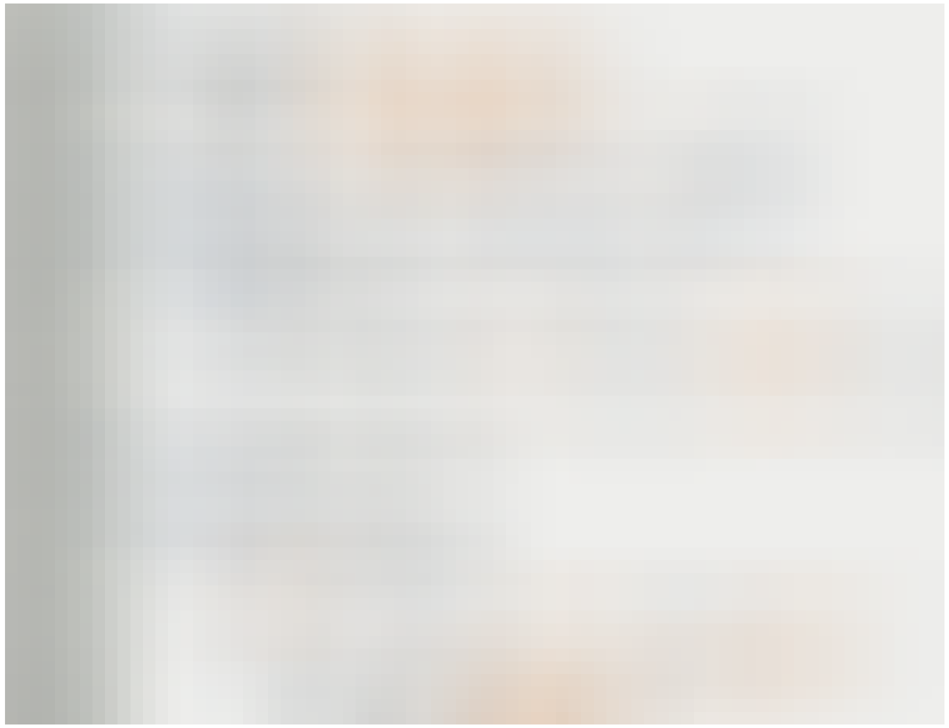
The first magical script

Yep, it turned out Chrome has a debugger API that even powers the debugger (the debugger is itself a bunch of HTML and browser Javascript that communicates with this API through Websocket). And Firefox and Safari and Edge and others have too, even though it’s not perfectly portable. Anyway, Chrome’s one works a bit better, so we’ll use Chrome’s. As matter of a fact, it’s even meant to replace most PhantomJS uses once a feature called “headless Chrome” will land.

The first part connecting to it is launching your usual Chrome, with the `--remote-debugging-port=<port number>` flag. Connecting to the specified port leads you to a HTTP server, on which you make a request to the “/json” endpoint that conveniently lists active tabs and Websocket URLs to connect to the debugger API for each. Then, you exchange JSON request/response/event messages on the socket in a simple way. A request will have a constant ID and map either to a response or error; events can be enabled using specific requests.

Some Websocket communication between Chrome and the Chrome developer tools (captured using Chrome developer tools themselves)

Let’s get our hands dirty: we’ll first call “Runtime.enable” in order to catch the “Runtime.executionContextCreated” event, that will fire when a page’s Javascript execution context is created; then, “Debugger.enable”, enabling the “Debugger.scriptParsed” event to fire when a script is loaded so we can look at it, and find functions that interests us to put breakpoints at these; “Network.enable”, so that we are notified about all HTTP requests in order to see which ones contain “!”-encoded data that we intercepted at Javascript level, and make a direct link between .proto definitions and HTTP endpoints; and finally “Page.navigate”, a simple way to tell “Browser, lead us to Google Maps!”.


Code snip: some commands sent at Websocket creation, and the begin of event loop

When a new script is seen, the “Debugger.scriptParsed” event is triggered. We then call “Debugger.pause” (in order to avoid any race condition) and

"Debugger.getScriptSource", that will give us ability to find our relevant function with a regex or string signature, and call "Debugger.setBreakpoint" to set a breakpoint, specifying script ID, line and column numbers.

Here is the Javascript function that we want to break in:

```
_.Eqa.prototype.H = function(a, b) {
    var c = Array(Fqa(a, b));
    Gqa(a, b, c, 0);
    return c.join("")
};
```

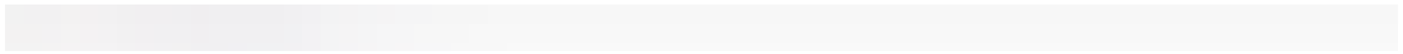
It takes two arguments: "b", a Javascript object that defines the structure of the Protobuf message; "a", another, simpler object that defines message data, that will be serialized using this structure; and returns "c", the raw "!"-separated data string.

"Fqa" calculates the size of the output array (4 * the number of fields, each field being made of four substrings that will be joined together: separator, field number, type character, field value), “Gqa” writes to it, String.prototype.join converts it to string.

So we’ll want to look at “b” and get it into a readable .proto, and remember “c” for later, so we can observe in which request it is sent. This means putting a breakpoint on the last line. Here’s what argument data looks like seen from the debugger:

Sample data for the ”a”, ”b” and ”c” local variables, respectively.

After some deduction, we conclude “b” structure’s would look like this:

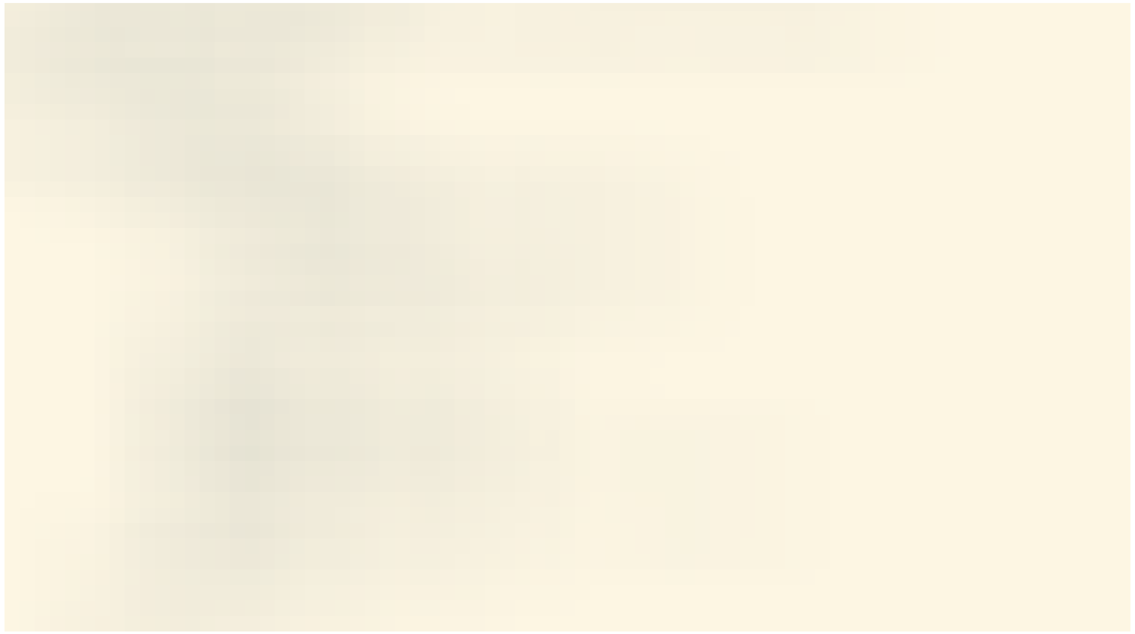


However, I have shown you a readable version, just figure the same with random letters instead of any text names and no comments. Most object attributes are minified (and change depending on the minifier run), so we'll have to use regexes on code to know which attributes correspond to what information.

We understand what this data means: next, generate the actual .proto. We could access directly variables and object attributes through the debugger API, however, this would be awfully slow so we can't really. We could try to convert it to JSON, but this will mostly not work, as nested messages can refer to their parent, and this would result in circular references (references to a parent object from its children). Third and right solution: just inject Javascript, let it do the work, return the string through any medium (I'll use the console API for reasons).

Javascript can be evaluated through the "Debugger.evaluateOnCallFrame" call locally, "Runtime.evaluate" call globally. Conveniently, you are notified of console messages through the "Runtime.consoleAPICalled" event. In addition to capturing network requests ("Network.requestWillBeSent" event), we'll also inject another bit of Javascript in order to hook the history.replaceState() API function, so we know when the main page changes its URL containing "!"-separated state data.

Some more code written, we launch the script controlling our Chrome instance, we conduct some Google Maps interaction to let it collect data, we now have the two elements we needed: reconstructed .proto structures, samples of URLs with the serialized data. The former will be written to disk in their regular form, the latter as convenient JSON with samples of requests put in arrays for every endpoint.


Some of the reconstructed .proto data

Some of the network endpoints, with captured data samples for each

Great. Now, what should we make of this?

The second nifty program

I don’t know if you like large control panels full of buttons, I bet you love picking randomly at these when you can. That’s what we’ll construct out of the dataset we cultivated. Protobuf messages are conveniently rendered as the GUI element called tree view; handy manipulation would be hard in mere command line.

Protobuf fields inferred out of the structures we seen have no names. We assigned one-letter names to them, as in minified code. We want to name them after what they actually do, for that, we’ll observe, and a convenient way to observe fast (for example, using your mouse wheel to mutate an integer field) would be great.

We also want to test all values, so basically, as we know what they do, we could for instance integrate reverse engineered structures elsewhere to build something brand new, or see whether some of them induce an interesting behaviour security-wise…

I like to use Python for everything, the first script used Python, the second one will use Python with Qt. The solution looks simple: using the QTreeWidget widget to render the message, each field will be a subclassed QTreeWidgetItem (with children widgets such as QLabel for text, QSpinBox for integers, etc.). The code will eventually get to look a little bit entangled, as we'll have to handle cases such as repeated fields (duplicating themselves on the UI when you fill one first), states of a field being checked, present in request, etc. But overall readable and way understandable.

The output page will be rendered live using QtWebEngine, the convenient Qt module that superseded QtWebKit and allows you to integrate Blink (in fact, a larger detached portion of Chromium) as a frame in your application. Also, a text area to display side information such as response hash, size, response code, mimetype.

So, you touch buttons, I want the response to render live, and slick experience, got it? Got it, after a few more hundred LoC, it's implemented and shiny.

The Protobuf editor, our second creation, a slick way to edit and replay network data we captured at the previous step.

The 0-day

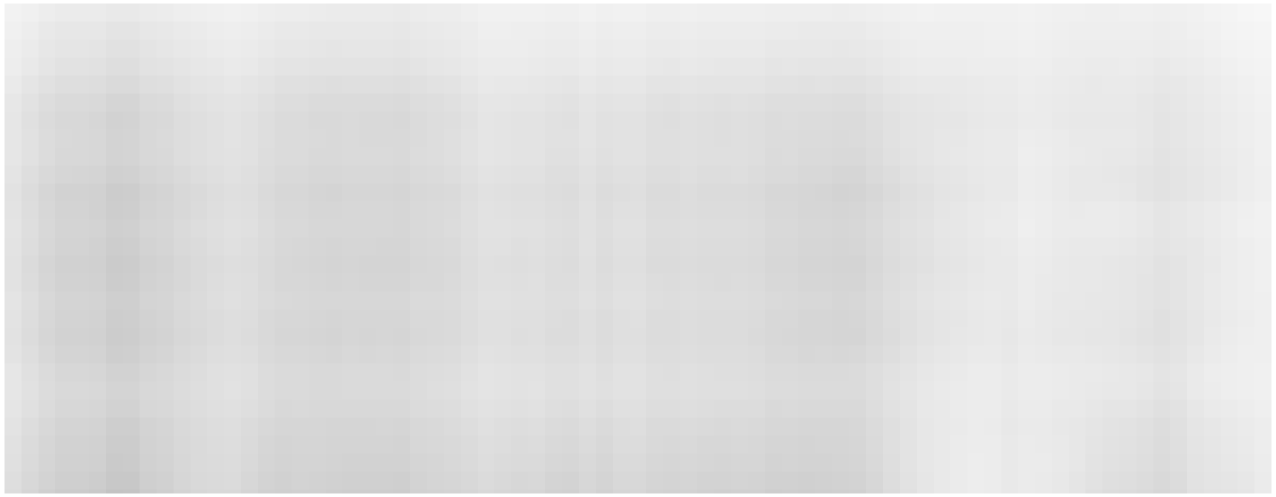
The whole request to the endpoint requesting a Google Maps tile (https://www.google.com/maps/vt) has about 730 fields, scattered around 125 messages. From a security point of view, that's great, because that's attack surface, and as much of these fields are implicit, hidden and required a reverse engineering work to get there, that's as much things untested from the public we can see.

Now I kept testing a lot of fields by hand, sometimes documenting what they do. Most of these didn't seem to do anything in the current setup I tried them. Eventually, I sorted out the important parts of the request.

First, the coordinates on the map. There are multiple ways to describe these, regular decimal WGS84, or a larger and more precise coordinates unit, useful for the satellite view, and there are also other ways to define a viewport, for example if you want a map of a given pixel size to cover a precise area. There's basically every way you would want to render a map. There are also all kinds of map layers, options to render labels on the map, position markers, draw an itinerary from a direction to another, cover all Maps features being drawn server-side, etc.

But more important, it doesn't only render bitmap. It also renders vector; the Android application used vector for years, the web more recently and for part of the requests. There are multiple formats for vector, all of these are proprietary.

The former was a regular binary format (signature "DRAT"), with binary structures convenient to what it's meant to store (map rendering data), and a version number to determine when a fields appears, changes meaning or disappears; data was obfuscated out of a simple RC4 key made out of tile coordinates, a fixed key and a nonce (I found a research paper (p. 71) briefly describing it, with a now outdated decryption key) and ZLIB, and is still in use by the Android application.

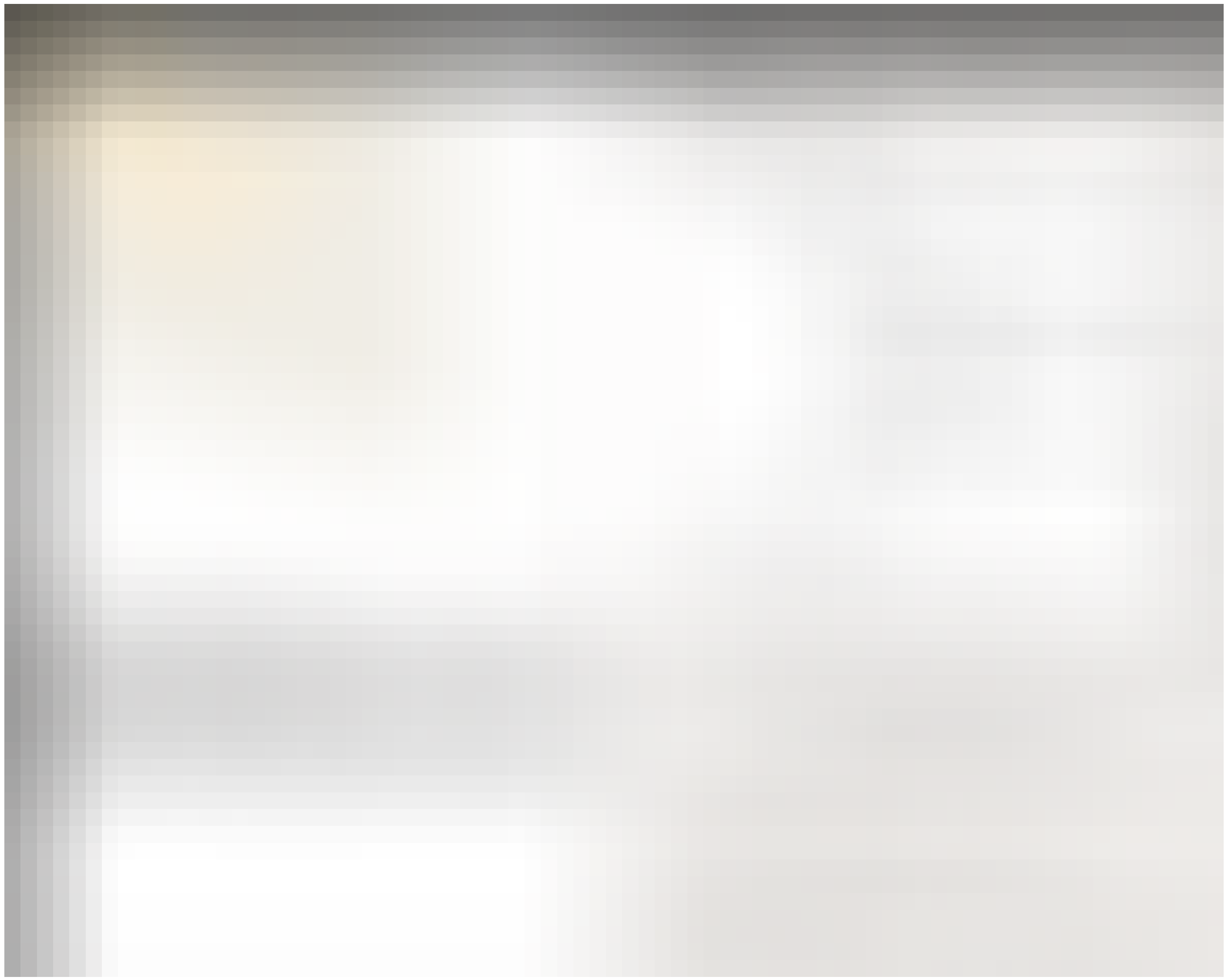


The proprietary "DRAT" format is used to transport maps in a vector format to the Android client.

The new one, integrated to the web application just uses binary Protobuf—again, that's a second form of Protobuf in the web app. A single-byte xor is used instead of RC4, what a ramp up in efficiency. That's the binary data sent through AJAX that we mentioned earlier, encapsulated in a mere length-value container format (signature "XHR1"). Transmitted data has mostly the same meaning as in the old (I made a small script to mostly render it to SVG just to spend time, it kind of works but would hardly be useful to a lot of people and could attract C&D's, so this is left as an exercise for the reader).

You can pick any of these formats versions, more than a dozen. One of these versions, specifically, was interesting. How? **It was output with a "Content-Type: text/html" header.** Yet, it was plain binary data. It was Protobuf-based like the new format, yet it had the header of the former one, so it had the same RC4+ZLIB processing. However, having a lot of options is interesting in security. And so there was a field that disabled encryption, and another one for compression. And you had raw map data, including the raw strings for places and markers, served as text/html to your browser.

What’s a cool way to inject arbitrary strings to the map? Placing markers! Editing other request fields does allow you to do that.



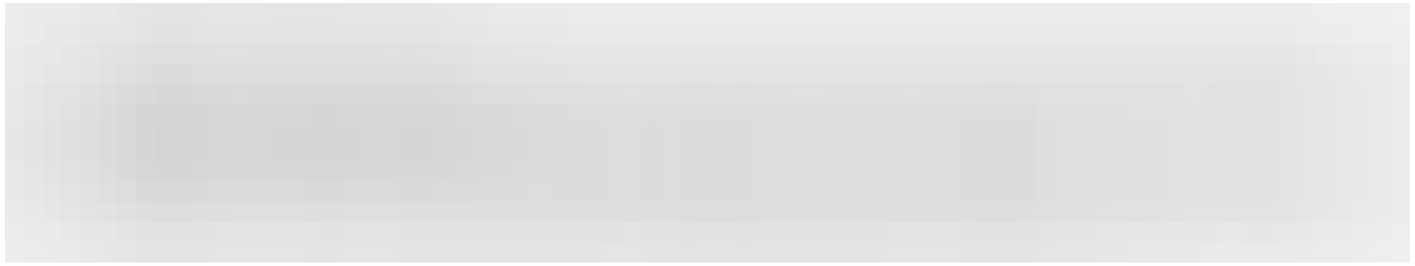
Placing a arbitrarily named marker on the map, by editing request information. On this screenshot, this is done while requesting map tiles in a raster format (PNG).

Once you try to do that while also setting fields relevant to enabling the undocumented format we found, the Javascript alert box valiantly shows up!

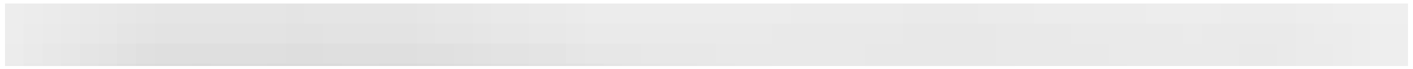
Our original XSS payload looked like this. 🎉

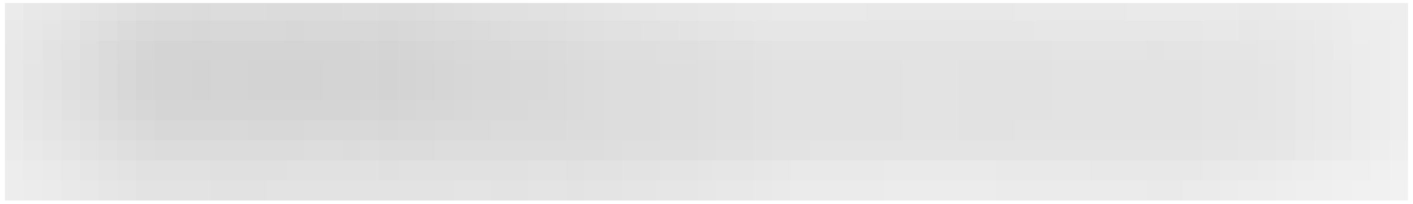
Let’s be even more clever and see what fields can be encoded without using any HTML-related chars. Browsers such as Chrome have filters to detect and block XSS when HTML is present both in request and response, that’s also the case of Firefox with extensions such as NoScript. Filter-escaping wit comes in here!

The “!”-separated data decoding routine accepts two data types corresponding to strings, even if it’s originally one Protobuf type. The “s” type character allows you to encode strings raw as part of the output string, only with “!” and “*” escaped as “*21” and “*2A” (the corresponding hex codes). The “z” character type allows you to encode the string as base64 (simple flavour, without padding). Know what, filter-based XSS defences are rendered useless.


The same payload, once the HTML string was base64-encoded.

And you can obfuscate even more, as it turned out the server converts on the fly the “!”-separated data to raw binary Protobuf, and most types are cast without check, and remember, strings and messages both share the same of the five data types in binary format. So, you can simply convert nested messages to binary, and pass them as strings, exposing even less what they do, and look more cool.

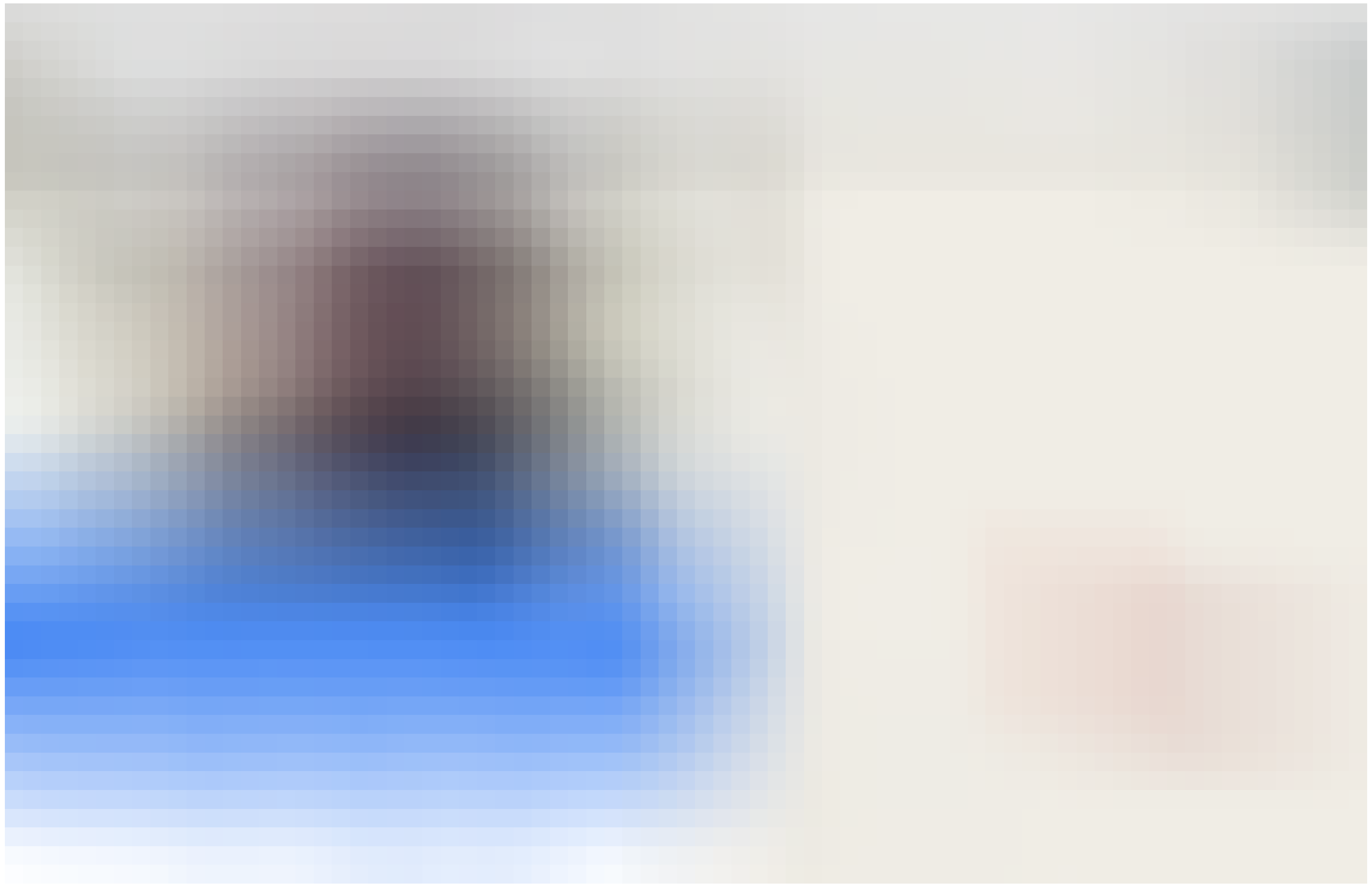


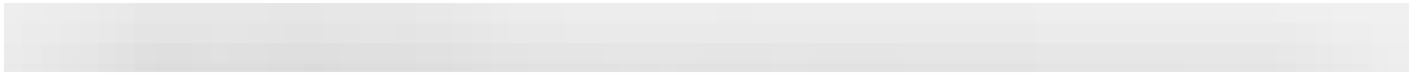

The same payload, once we converted nested messages to wire format, then cast to base64 string—now that's unintelligible.

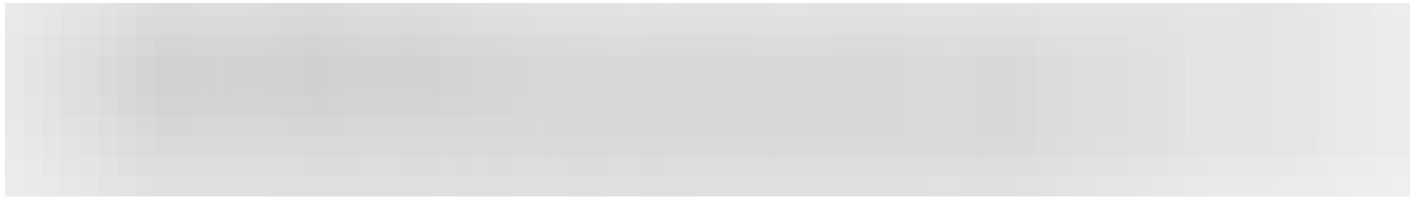
Now we have a XSS payload that will work with anything—Chrome, Firefox, Safari, whatever accepts Javascript—on the www.google.com domain, as the former maps.google.com was dropped to put back the service on the main domain…

Just for the sake of cleverness, let's find another way. It turns out anyone can submit new places to Google Maps, that may appear on the map with more or less moderation (a posteriori). Did someone submit a <script>alert(1)

</script> place to the map already? Sure, some Chinese and Indian guys did, one even showing off its hacker face of dread!


With Google Maps, find nearby hot hackers from your area





A payload leveraging an existing place on the map.

We have four exciting ways to exploit this neat 0-day—this looks enough. Let's move to the next step.

Vendor response

Google provides standardized rewards for most security bugs, as described on the page for its vulnerability reward program. In our case, a proper XSS on the www.google.com domain will usually trigger a $5,000 bounty.

Day 0—I fill in the report form.
Day 0.54—My report is triaged, already!
Day 0.64—My report is validated, bug filed.
Day 1 (or so)—Well, this is fixed.
Day 1.84 —

You know what, great bounty gets turned into motivation, and great motivation gets turned in open-source.

The project

The code linked to this research since got a bit more substantial: I looked around for more, and seen that a lot of applications on the Android platform (including Maps') use Protobuf, and it seemed to have a lack of an unified tool for extracting these structures (a few localized scripts existed but targeted specific apps or implementation flavours, and mostly didn't support

Proguard). And desktop apps, mostly in C++ existed in addition to mobile Java stuff.

So I started to glue together scripts I wrote on a few occasions, put a same GUI behind, added a few features, and it started to look like something.

You can find the current state of the project that departed from this at **https://github.com/marin-m/pbtk**. Here are the main parts of the code discussed above:

- extractors/web_extract.py: that's the main extraction .proto script, that launches Chrome, connects to its depths and does the stunning work. You can use it individually, just pass an URL as an argument.
- views/fuzzer.py: that's the little GUI editing and replaying requests. It's now integrated to the larger app, and that you can play with launching the GUI (gui.py) and clicking "Step 3…". It will look for .protos stored in a folder of you home directory, as explained in the interface.
- In addition, the code required to encode back and forth URL parameters in the "!"-separated format is bundled as a module in utils/pburl_decoder.py. It is included by the previous.

That's the end of this note—hope you appreciated it, and see you later for more reverse engineering adventures!

Programming Reverse Engineering Protobuf Python Security

Like what you read? Give Marin Moulinier a round of applause.

From a quick cheer to a standing ovation, clap to show how much you enjoyed this story.

1.1K 12

Marin Moulinier

https://github.com/marin-m

Follow

Also tagged Protobuf

Retrofit advance: Multi-Converter

Jintin

3 min read 142

Also tagged Security

How I Hacked 40 Websites in 7 minutes

Georgios Konsta...

9 min read 108K

Also tagged Programming

How you can improve your workflow using the JavaScript...

Riccardo Canella

5 min read 1.4K

Responses

Write a response…

Show all responses