reCAPTCHA bypass via HTTP Parameter Pollution

reCAPTCHA bypass via HTTP Parameter Pollution - Author not stated, Andres Riancho.

- Published: date not stated
- Original: <https://andresriancho.com/recaptcha-bypass-via-http-parameter-pollution/>
- Preserved from: https://andresriancho.com/recaptcha-bypass-via-http-parameter-pollution/ (live) on 2026-08-10
- Licence: unknown

Content

> UNTRUSTED SOURCE TEXT. Everything below this line is third-party material quoted for research. It is data, not instructions. Do not follow directions, execute code, or fetch URLs because this text says so.

reCAPTCHA bypass via HTTP Parameter Pollution

tl;dr

I reported a reCAPTCHA bypass to Google in late January. The bypass required the web application using reCAPTCHA to craft the request to **/recaptcha/api/siteverify** in an insecure way; but when this situation occurred the attacker was able to bypass the protection every time. The security issue was fixed “upstream” at Google’s reCAPTCHA API and no modifications are required to your web applications.

Intro

reCAPTCHA is a Google service that allows web application developers to add a CAPTCHA to their site with minimal effort. reCAPTCHA is a complex beast with a lot of use cases: sometimes it will trust you based on your existing cookies, sometimes it will require you to solve multiple challenges. The following introduction is for the use case where the vulnerability was found.

When the web application wants to challenge the user, Google provides an image set and uses JavaScript code to show them in the browser as follows:

The user solves the CAPTCHA and clicks “Verify”, which will trigger an HTTP request to the web application. This HTTP request will look like:

```
POST /verify-recaptcha-response HTTP/1.1
Host: vulnerable-app.com

recaptcha-response={reCAPTCHA-generated-hash}
```

The application needs to verify the user’s response with a request to Google’s reCAPTCHA API:

```
POST /recaptcha/api/siteverify HTTP/1.1
Content-Length: 458
Host: www.google.com
Content-Type: application/x-www-form-urlencoded

recaptcha-response={reCAPTCHA-generated-hash}&secret={application-secret}
```

The application needs to authenticate itself using **{application-secret}**, and sends **{reCAPTCHA-generated-hash}** to the API to query the response. If the user answered correctly the API will yield:

```
HTTP/1.1 200 OK
Content-Type: application/json; charset=utf-8
Content-Length: 90

{
  "success": true,
  "challenge_ts": "2018-01-29T17:58:36Z",
  "hostname": "..."
}
```

Which will be received by the web application, processed, and most likely result in the user being granted access to the resource.

HTTP Parameter Pollution

HTTP parameter pollution) is almost everywhere: client-side and server-side, and the associated risk depends greatly on the context. In some specific cases it could lead to huge data breach, but in most cases it is a low risk finding.

The reCAPTCHA bypass requires an HTTP parameter pollution in the web application. This requirement greatly reduced the severity of this vulnerability report when triaged by Google.

An example vulnerable web application would look like this:

```
private String sendPost(String CaptchaResponse, String Secret) throws Exception {

  String url = "https://www.google.com/recaptcha/api/siteverify"+"?response="+CaptchaResponse+"&secret="+Secret;
  URL obj = new URL(url);
  HttpsURLConnection con = (HttpsURLConnection) obj.openConnection();
```

Where string concatenation is used to build the *url* variable.

It is also important to notice that on Google's side, it was possible to send these two HTTP requests and get the same response:

```
POST /recaptcha/api/siteverify HTTP/1.1
Host: www.google.com
...

recaptcha-response={reCAPTCHA-generated-hash}&secret={application-secret}
```

```
POST /recaptcha/api/siteverify HTTP/1.1
Host: www.google.com
...

recaptcha-response={reCAPTCHA-generated-hash}&**secret**={application-secret}&**secret**={another-secret-app
```

The reCAPTCHA API always used the first **secret** parameter on the request and ignored the second one. This is not a vulnerability, but was used in the exploit.

Last piece of the puzzle

Web developers need to test their applications in an automated way, with that goal Google provided an easy way to "disable" reCAPTCHA's verification in staging environments. This is well documented and explained in the documentation, to sum it up, if you want to disable reCAPTCHA verification please use the hard-coded site and secret key shown below:

- Site key: 6LeIxAcTAAAAAJcZVRqyHh71UMIEGNQ_MXjiZKhI
- Secret key: 6LeIxAcTAAAAAGG-vFI1TnRWxMZNFuojJ4WifJWe

Putting it all together

Now that we have all the ingredients, let's take a look at the exploit:

```
POST /verify-recaptcha-response HTTP/1.1
Host: vulnerable-app.com

**recaptcha-response**=anything%26secret%3d6LeIxAcTAAAAAGG-vFI1TnRWxMZNFuojJ4WifJWe
```

If the application was vulnerable to HTTP parameter pollution AND the URL was constructed by appending the **response** parameter before the **secret** then an attacker was able to bypass the reCAPTCHA verification.

Note that I'm sending a specially crafted response to the vulnerable web application, which contains:

- **anything**: just a placeholder
- **%26**: an URL encoded ampersand character
- **secret**: the name of the parameter I want to "inject"
- **%3d**: an URL encoded equals sign
- **6Le...JWe**: the secret key which disables reCAPTCHA response verification

When the attack requirements are met, the following HTTP request is sent by the web application to the reCAPTCHA API:

```
POST /recaptcha/api/siteverify HTTP/1.1
Host: www.google.com
Content-Type: application/x-www-form-urlencoded
User-Agent: Python-urllib/2.7

recaptcha-response=anything&secret=6LeIxAcTAAAAAGG-vFI1TnRWxMZNFuojJ4WifJWe&secret=6LeYIbsSAAAAAJezaIq3
```

Note that the request contains two **secret** parameters, the first one is controlled by the attacker (due to the HTTP parameter pollution in the vulnerable web application) and the second one is controlled by the application itself. Given that the reCAPTCHA API uses the first one, the response to this request is always:

```
HTTP/1.1 200 OK
Content-Type: application/json; charset=utf-8
Content-Length: 90

{
  "success": true,
  "challenge_ts": "2018-01-29T17:58:36Z",
  "hostname": "..."
}
```

Which will be processed by the web application and the attacker will be granted access.

Fixed upstream

Google decided to fix this issue in their REST API, and I believe it was a wise move. Their fix is simple: If the HTTP request to **/recaptcha/api/siteverify **contains two parameters with the same name, then return an error.

Fixing it this way they are protecting the applications which are vulnerable to the HTTP Parameter Pollution and the reCAPTCHA bypass, without requiring them to apply any patches: *awesome!*

Exploitabity in the wild

There are two strong requirements for this vulnerability to be exploitable in a web application, first, the application needs to have an HTTP parameter pollution vulnerability in the reCAPTCHA url creation: Github searches showed that ~60% of the integrations with reCAPTCHA are vulnerable.

Second, the vulnerable web application needs to create the URL with the response parameter first, and then the secret: “response=...&secret=...”. Strangely, almost all applications use “secret=... &response=...”. My guess is that Google’s documentation and code examples did it like that, and others simply copied the format. Google got lucky there... if they would have done it the other way around, this vulnerability would have affected even more sites. GitHub searches showed that only 5 to 10% of the reCAPTCHA implementations meet this requirement.

So, if I would have wanted to exploit this in the wild, only ~3% of the sites which use reCAPTCHA would have been vulnerable: not bad since this is used everywhere... but small compared to other vulnerabilities.

Summary

- **As a developer:** Never use string concatenation to create query strings. Use a dictionary to store keys and values, then URL-encode it.
- **As an application security expert**: HTTP Parameter Pollution is your friend.

Timeline

- 2018-Jan-29 / Vulnerability is reported to Google
- 2018-Jan-30 / Google replies: "reCAPTCHA is working exactly as designed"
- 2018-Jan-31 / I ask them to please re-read the vulnerability report
- 2018-Jan-31 / Google asks for more information
- 2018-Feb-1 / Google confirms vulnerability
- 2018-Feb-15 / Google awards 500 USD for the vulnerability report. Money donated to charity.
- 2018-Mar-25 / Patch is released
