

When Good Services Go Wild: Reassembling Web Services for Unintended Purposes

Feng Lu, Jiaqi Zhang, Stefan Savage

University of California, San Diego

Abstract

The rich nature of modern Web services and the emerging “mash-up” programming model, make it difficult to predict the potential interactions and usage scenarios that can emerge. Moreover, while the potential security implications for individual client browsers have been widely internalized (e.g., XSS, CSRF, etc.) there is less appreciation of the risks posed in the other direction—of user abuse on Web service providers. In particular, we argue that Web services and pieces of services can be easily combined to create entirely new capabilities that may themselves be at odds with the security policies that providers (or the Internet community at large) desire to enforce. As a proof-of-concept we demonstrate a fully-functioning Web proxy service called CloudProxy. Constructed entirely out of pieces of unrelated Google and Facebook functionality, CloudProxy effectively launders a user’s connection through these provider’s resources.

1 Introduction

The modern Web service ecosystem is one built on composition; using Web-based server APIs and client Javascript to create new services and capabilities. With a few lines of code one can connect a Google Maps widget to a Facebook app with a Microsoft datasource and, in so doing, “mash-up” an entirely new Web service neither envisioned, nor endorsed, by any of the individual service providers. Overall, this programming environment promotes reuse and agility, but inevitably at the expense of encapsulation or clean semantic guarantees. It is no surprise then that this dynamic style of programming can create new security risks; cross-site scripting, cross-site request forgery and so on. However, to date most of the attention on these problems has focused on violations of the *client’s* security policy—what can a Web site do to a browser? [10, 13, 14].

In this paper, we opine that there is another class of security concerns that involve the lack and difficulty of policy enforcement by the *providers* of Web services. As a trivial example, Google’s GMail was designed and intended as a free e-mail service, but systems such as `gmailfs`[4] bypass this intent by wrapping the APIs to create a free file system service. Similarly, the Graffiti network, abusively implements file storage using blog spam on open forums [11].

However, these simple examples belie the potential

complexity that can arise from exploiting *combinations* of services, both within and across providers. Many modern Web services can have both local and remote side-effects, can fetch objects from other sites, can change local state, and can invoke additional services encapsulated as arguments. In this manner, a user may leverage the reputation of a collection of Web service providers to engage with a third-party on their behalf.

In this paper, we explore this issue by example, demonstrating the creation of a functioning Web proxy from unrelated components. In a manner metaphorically similar to Shacham’s “return-oriented programming” [12] we demonstrate how pieces of correctly-functioning Web services from different providers can be stitched together to create completely new functionality. The resulting free service, CloudProxy, launders a user’s connection through the servers of Google and Facebook. This capability could be used to bypass IP-level access restriction (e.g., such as commonly used to restrict streaming video or purchasing options within geo-locked regions), to commit mass Web spam without fear of blacklisting (i.e., no Web site depending on advertising can afford to blacklist Google IPs since Googlebot visits are what fills the Google search index), or to mount denial-of-service attacks on third parties using these provider’s resources (e.g., by repeatedly downloading large objects). In the remainder of this paper we detail the design process for building our CloudProxy service and some of these risks it creates. However, our service is less a threat in and of itself than a proof-of-concept that abusive Web service composition can be used to synthesize new threats from benign pieces. Moreover, since these threats can be inherently cross-domain, there are interesting new challenges for how to best address such problems.

2 Design Overview

In designing our Web proxy, we focused our efforts on the most widely used HTTP methods: GET and POST.¹ Our design approach is summarized as follows:

- We make use of public Web service APIs that have, either explicitly or implicitly, core functionality that allows us to retrieve content given an URL,

¹Note that while other HTTP methods can be implemented in a similar way, we omit the details due to space constraints.

- If necessary, each request is rewritten to meet the constraints of these Web service APIs; this pre-processing step is itself accomplished by exploiting (other) existing Web services,
- The responses from these Web services are reassembled in order to make the final response transparent to the user’s Web browser;

While this approach is inelegant at times (made so in particular by our need to transform arguments to meet API restrictions or requirements) we will show that it is sufficient to construct a fully functional Web proxy, capable of handling the majority of requests that might be made by a Web browser (e.g., page views, forms, real-time video streaming, etc.)

In the remainder of this section, we explain in general how Web service APIs are repurposed, how we find appropriate APIs and how we handle per-API constraints.

2.1 The Building Blocks of Our Implementation

Unlike normal Web servers, which implement HTTP commands directly, the building blocks for our implementation are the APIs provided by third-party Web services such as Facebook or Google. A typical such API can be a well-documented function call. For instance, it might be provided in a public library that is written in any language such as Python or Java (e.g., the Google Document API[3]).

In our design, we make use of APIs that themselves perform HTTP commands; such building blocks that widely exist in current Web services. For example, consider the scenario in which a user wishes to provide some input data to an online document processing service. If the data is available on the Internet (e.g., such as via a Web server), rather than requiring the user to download the data to local storage and then upload it again, document service APIs typically provide an interface for users to specify the Internet location of the content, which is then downloaded directly to the service provider. This “loads for” capability, suggests that such document service APIs may provide a likely base “gadget” for building a Web retrieval interface.

2.2 Discovering the Useful APIs

The APIs we target can be provided explicitly as the major functions of a popular Web service (e.g., such as a URL shortening service), or implicitly as an ingredient of such an original service. In the former case, the work of finding the APIs devolves to enumerating the set of services and their attendant side-effects (e.g., as with the document API example) to find an appropriate set of candidates. On the other hand, if the needed API is not pro-

vided explicitly, or is provided with undesirable restrictions, additional transformation may be required to provide the desired functionality. For example, the Google Spreadsheet service supports importing images and displaying them on the Web. However, if one wants to fetch an individual image using this service, it may require decomposing the service into its constituent parts to identify the key unbundled piece of functionality.

Given the limited structure imposed on Web APIs, we do not currently have a good mechanism for systematically and automatically detecting all possible building blocks. However, we observe that our own modest knowledge combined with some manual investigation has been sufficient for the goals of this paper. As major providers regularize their APIs into well-formed catalogs this process will only become easier and we further believe that implicit APIs will be susceptible to automated discovery and enumeration through techniques such as program analysis of Web service Javascript.

2.3 Adjusting the Input and Output

Finally, standard HTTP requests can include URLs and other parameters, which are represented as strings. However, these strings sometimes may not meet the input requirements of the APIs we wish to use. In this case, we need to craft the request string so that it is recognizable by the target APIs. There is a similar problem on output. Depending on the API we use, the output may be transformed to match the formatting rules of the API’s specifications. For these situations, we must implement the reverse transformation and normalize the content representation between service interfaces. Surprisingly, we have found that even these transformations can be performed entirely using third-party Web service APIs (detailed in Section 3).

3 Implementation

In this section, we detail the concrete steps involved in making our proxy implementation fully functional. To be specific, our goal is to reimplement the functionality of HTTP GET and POST as well as obtaining the final URL of a Web object. We next explain how we implement each piece of functionality in turn, including identifying its requirements, finding appropriate service APIs and, when necessary, how services are composed to work around per-API restrictions.

3.1 HTTP GET

Retrieving Web content is typically accomplished with HTTP GET method. We first describe how regular ASCII HTML content is fetched and then discuss how non-ASCII content requires its own work-arounds.

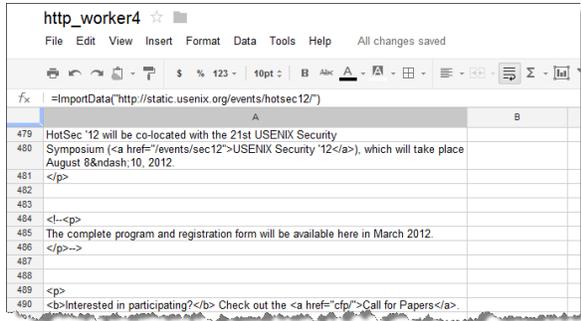


Figure 1: **Importing Webpage in Spreadsheet.** When a Webpage is loaded into Google Spreadsheet, the raw content of the HTML page is displayed.

3.1.1 ASCII Based Content

As discussed in the design section, most Web services aim to process all data inside the cloud and provide only the final results to end users. However, there is a fundamental similarity between a service retrieving online Web objects for cloud processing and a browser retrieving online Web objects for display — both require a full implementation of HTTP GET.

The first such API that came into our attention is the *ImportData(.)* function provided by Google Spreadsheet. It is designed for users to quickly populate their spreadsheets based on online CSV or TSV files. This function takes a single parameter—the URL of the CSV or TSV file. Interestingly, we notice that this function can be used to retrieve any Web object, not only spreadsheets. Figure 1 shows a screenshot of retrieving a Webpage in spreadsheet.

Unfortunately, as Figure 1 shows, Web content fetched via this interface is split across multiple spreadsheet cells as opposed to staying in the same cell. In particular, the newline character ‘\n’ splits the content across rows, while the comma ‘,’ splits content across multiple columns in the same row. Thus, we must reconstruct the original content by reversing this process; we scan cells from the first row and for each row, we start from the first column, adding newline characters and commas accordingly.

Since most Web browsers will request multiple Web objects simultaneously, we use multiple spreadsheets as workers and dynamically assign retrieval requests to available workers. In our implementation, we create four Google Doc accounts, with each account hosting ten spreadsheets. Thus, at any point in time, forty spreadsheets are available for web content download. For the web tasks we performed, forty workers seem to be adequate. One could always add more workers by creating more spreadsheets or Google Doc accounts (exploiting the fact that these cloud services are essentially free of

charge).

However, one limitation we identified in using the *importData(.)* function is that it only supports ASCII based contents (not surprising since binary data is not evaluable in a spreadsheet cell). Unfortunately, not all Web content contain pure ASCII characters, and this is especially true for multimedia files such as images, sounds or video. Thus, to support non-ASCII content, we must turn to alternative APIs.

3.1.2 Non-ASCII Content

Even though the *ImportData(.)* function does not support binary data, Google spreadsheet and other document types (e.g., document, presentation) do support embedded images. For example, in the Google document GUI, users can specify the URL of an image they want to import into their documents, and the corresponding image will appear in the document once the URL is submitted. Inspecting the corresponding Javascript code, we observed that the Google cloud downloads the image, stores it in Google’s server’s and then assigns a new URL to the image. It is this new URL that is then passed to Google documents (the same approach is shared by Google presentation as well). Unfortunately, the Google document/presentation service does not export APIs for image insertions.

However, returning to Google spreadsheets, we found that images are handled quite differently. In particular, a content server API function call is used to retrieve the image with the original image URL as part of the function parameters.² Even though there are many parameters in the content server API invocation example found in the spreadsheet (Figure 2), only two parameters are compulsory: ‘URL’ and ‘container’.³ Figure 2 gives the original API call example found in spreadsheet as well as our simplified version. Further exploration on the Google content server API suggests that not only images are supported but in general all UTF-8 encoded contents are handled as well.

3.2 HTTP Redirect

Finally, if the URL of a requested web object has changed, the Web server typically returns a HTTP redirect or location update message with the new URL embedded. On seeing such a redirect message, the Web browser simply makes another request and updates the browser address bar accordingly. In practice, to reduce the number of URL changes that a Web server has to remember, the locations of many Web objects are specified

²Note that “content server API” is our own name for this service. We have been able to find any documentation for this functionality nor a public name for its API.

³Eventually we discovered that the actual value of parameter ‘container’ does not play any role, and the mere presence of this parameter is sufficient.

<p>Full API: images-docs-opensocial.googleusercontent.com/gadgets/proxy?url=http%3A%2F%2Fopencage.info%2Fpics%2Ffiles%2F800_13785.jpg&container=docs&gadget=a&rewriteMime=image%2F*&resize_h=800&resize_w=800</p>
<p>Essential Parameters: images-docs-opensocial.googleusercontent.com/gadgets/proxy?url=xxx&container=###</p>

Figure 2: **API Invocation Examples.** The upper example shows the original format of the API; we show the simplified version in the bottom, where only `url` and `container` are necessary parameters.

as relative paths with reference to some known Web objects.

During our implementation for the HTTP GET method, we realized that Google APIs hide such HTTP redirect messages when fetching Web contents. In other words, the Google cloud service will automatically follow the updated location and download the final Web object. We consider this a rational design for cloud service providers as users do not need to know such details and only the final outcome matters. However, not being able to observe HTTP redirect messages proves to be a hurdle for us. In particular, a Web object can contain references to other objects. For example, a Web page may include an image and the location is specified by a relative path based on the current location of the Web page. If the original URL of the Web page is not the same as the final URL (i.e., due to redirects), it is impossible for us to obtain get the URL for the image object. This is exacerbated because Web servers do not routinely redirect requests for Web objects specified by relative path names and thus accesses to embedded objects relative to the old URL will fail.

Given these limitations, we must find another mechanism to check whether the requested URL has been changed. If changed, we must also obtain the final URL location and update the client Web browser accordingly. In service to this goal, we observed that Facebook has a debug API which allows developers to enter the URL of their open graph protocol [2] based Web page, and obtain a variety of high level meta-data about the page. In fact, the Facebook debug API is able to return this information about a URL regardless of whether the corresponding Web page contains open graph protocol or not. Since the HTTP 301/302 redirect information is part of the meta-data returned, we were able to use this approach to first query each URL before fetching data from its final location using our GET implementation.

3.3 HTTP Post

While data can be delivered to a Web server via the HTTP GET method (i.e., using URL-encoding), HTTP POST is generally the preferred mechanism for sending data. In particular, POST is not limited the amount or structure of data to be sent and is harder to abuse via encoding attacks[7]. However supporting the HTTP POST method within CloudProxy is not a simple extension to our GET implementation since neither the Google spreadsheet or content server API use POST. Instead, in searching the available Google service, we discovered the Google gadgets caching API, which is designed to cache gadgets on the Google cloud in order to speed up subsequent gadget access. This API allows the caller to issue an HTTP POST request, NOT to Google but to the actual URL as specified in the API function call (i.e., the location of the gadget to be cached).

3.4 API Friendly URL Design

So far, all of our work assumes that URLs can interpreted. Unfortunately, this is not also the case. While our experience is that the Facebook APIs we used were robust across URL encoding, several of the Google APIs were highly sensitive to the form of URL. In particular URLs wither special syntax characters such as space, comma(','), and ampersand('&') or their encoded equivalents (e.g., a space might be encoded as '%20') are not handled well by Google's content server or gadget caching APIs we use and thus these calls fail on such URLs. Thus, we need to convert all URLs into a normalized that is supported by all of our APIs.

To address this problem, we repurpose URL shortening as a form of normalization. URL shortening is a technique by which a service represents a URL in a compact manner and subsequently expands it using associated state. For example, the URL "http://bit.ly/aUAdWi" when presented to bit.ly will redirect to "http://www.cs.ucsd.edu/". URL shorteners are popular in messaging services that limit the number of characters in a message (e.g., such as Twitter). Since the output is typically only consists of 26 letters (lower and upper cases) and 10 numbers, it is ideal for our purpose of normalization. In our implementation, we used the Google URL shortener service, and the encoded URLs are passed to the various APIs.

3.5 Invocation Sequence for a Proxy

We combined these techniques to build a full functional Web proxy. Figure 3 shows the underlying workflow of the proxy.

First, when a user submits a URL, the proxy consults the Facebook debug API to obtain the final URL. If there is a redirect, we will construct a HTTP redirect response and send it back to the browser. With the

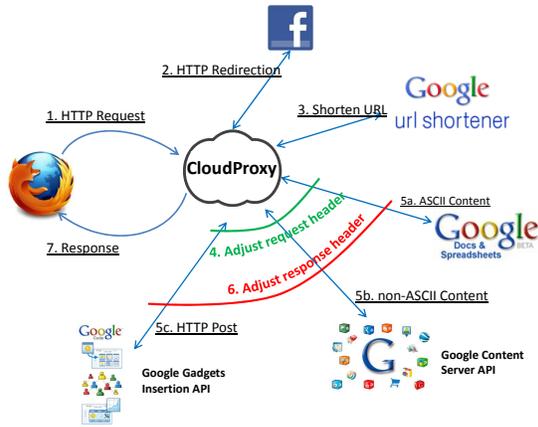


Figure 3: **Detailed Architecture Design.** The numbers indicate the execution order of each step. Depending on the requested content and type, we may choose one service from 5.a, 5.b, and 5.c

correct URL, we then invoke the Google URL shortener service to obtain an API friendly representation of the URL. If the requested content is ASCII based (determined via the MIME type in the HTTP response header), the Google spreadsheet API will be called. Otherwise, the Google content server API is used to obtain UTF-8 encoded non-ASCII Web contents. If the browser issues an HTTP POST request, we forward it to the Google gadget caching API. When a response is returned, we construct an appropriate HTTP response message and send it back to browser. Inside the HTTP response header, we purposely specify no-cache in the cache-control field since we have no idea about the cache validity period. In our actual implementation, we also introduce a number of optimization techniques to reduce processing overhead. For example, we do not check the URL for a possible redirect if it is constructed based on relative path.

Note that we do not believe our implementation is the only or even best implementation of this functionality. With the growth of Web services, there are a tremendous number of documented and undocumented APIs that could be exploited, sometimes with equivalent capabilities (there are tens of URL shorteners) and sometimes with different strengths and weaknesses. For example, we suspect that much of our functionality could have been replicated using the online document processing services of other providers equally well [8, 6].

4 Evaluation

We implemented the HTTP GET and POST commands and built the proxy glue using Python. To test our implementation, we instantiated the CloudProxy service and configured our Web browser to use it for all Web protocols based on HTTP requests. We tested a range of Web

Web Task	Video Demo Link
HTTP POST	http://youtu.be/74ari2Cgb9k
IP hiding	http://youtu.be/Rg2x_dbSNuI
HTTP redirect	http://youtu.be/8f82n-R-zsI
Video demo	http://youtu.be/PTZptNdOg-A
Spreadsheet demo	http://youtu.be/vKn8Gg0w10
Bing search demo	http://youtu.be/vEGG1wT5Evs

Table 1: Video Demo Links

IP	Host Name
66.220.149.53	star-11-02-snc5.facebook.com
66.220.153.28	star-11-03-ash2.facebook.com
69.171.224.29	star-12-01-prn1.facebook.com
173.194.64.95	oa-in-f95.1e100.net
74.125.224.202	lax02s02-in-f10.1e100.net
74.125.224.204	lax02s02-in-f12.1e100.net
74.125.224.211	lax02s02-in-f19.1e100.net

Table 2: Reverse DNS Lookup for Captured Source IP Addresses

sites, from simple static web pages to complicated interactive Web applications. Table 1⁴ lists the Web tasks we performed and the corresponding screen recording video for that task.

In order to confirm that there is no traffic between our local machine and the Website visited, we used Wireshark to record all incoming and outgoing packets to a Web server under our control. For each of the source IPs identified by Wireshark, we perform a reverse DNS look up to obtain its corresponding host name. Table 2 displays the results for one of the several Web tasks performed (the video example). All of these hosts are either owned by Facebook or Google, and the results indeed confirm that our proxy does not connect to Web servers directly.

Finally, even though CloudProxy can perform many useful Web tasks, there are two limitations that we need to address in the future. The first is that cookies are not supported in our current version. All of the cloud APIs we utilized ignore the cookie values sent by our proxy. Similarly on the reverse direction, no cookie data is returned from the cloud API invocation results. The other limitation is that the largest Web object size supported by the APIs we used is exactly 10 Megabytes. Any single response with size greater than this value will be discarded by the APIs. We believe that the size limitation is not a fundamental flaw of our approach, rather it is due to the particular content server API we employed. The Google content server API is intended to support images,

⁴Note, the name “Larry Lau” that appeared in these videos is an alias solely created for the purpose of this work.

the 10Mbytes threshold will not be a problem for most images provided by their customers.

5 Security Implications

Proxies, by their nature, have security implications since they remove a layer of information about the user. Sometimes this level of indirection can offer benefits (e.g., anonymity) while in other cases the same property provides opportunities for abuse (attacks with impunity). In this section, we review some of the direct security implications of a free proxy service such as we have described, from the viewpoint of different parties:

Web Content Providers: IP-based geo-restriction is a popular way to prevent certain users from accessing online resources [1]. It is frequently employed by Internet content providers to ensure that only users from certain geographical regions are able to view their contents (or, with e-commerce sites, to restrict who may purchase an item). The technical basis for such restriction is IP-based geo-location services [5], which implicitly depend on the source IP of the user being from their country of residence. Thus, using a system like CloudProxy, users can effectively bypass the aforementioned restrictions by exploiting Web service replicas in the country of interest.

End Users: CloudProxy makes it possible for entirely anonymous Web access using services that are, on their face, unrelated to such a capability (unlike dedicated proxy services such as Tor [9]). Using the service, a user's IP footprint would always appear to originate from Google or Facebook IP blocks.

Black Hats: The very fact that CloudProxy's requests are satisfied by Google and Facebook provides a resource asymmetry that attacks can exploit. Google, for example, has tremendous bandwidth and server capacity available to it that could be brought to bear on innocent third-parties. For example, during the implementation of CloudProxy, we noticed that the HTTP range field is only respected between our proxy and Google. However, Google servers will still retrieve the full content requested from a third-party Web server. This asymmetry allows a nefarious CloudProxy user to mount a denial-of-service attack by repeatedly requesting small portions of large files from a third party site, letting Google and the site manage the bandwidth of the full download.

Web Service Providers: Finally, while Google and Facebook have prodigious resources, most services are designed around a particular usage mode and may not operate efficiently in a different regime. For example, when Web contents are retrieved using Google gadget cache API, a copy of the Web content will be retained. If there is a substantially large user population accessing web through CloudProxy, this could quickly create undesirable resource demands.

6 Conclusion

In this paper, we argued that Web services, or pieces of Web services, could be easily combined to create new capabilities that may largely deviate from the service providers' original intention. As a proof of concept, we demonstrated that a web service based proxy, namely CloudPROxy, can be constructed from Google and Facebook's APIs without much difficulty. Our evaluation suggested that CloudProxy is able to accomplish most typical Web tasks. Furthermore, we discussed the direct security implications of such a proxy imposed on content providers, end users and web service providers. We believe that these security concerns are rooted from the lack and difficulty of policy enforcement by the providers of Web services, a research topic which deserves attention going forward.

References

- [1] Dynamic ip restrictions. <http://www.iis.net/download/dynamiciprestrictions>.
- [2] Facebook open graph protocol. <https://developers.facebook.com/docs/opengraph/>.
- [3] Google document api. <https://developers.google.com/google-apps/documents-list/>.
- [4] How to convert gmail into 8 gb of free remote storage. <http://pcmichiana.com/how-to-convert-gmail-into-8-gb-of-free-remote-storage/>.
- [5] Ip to geo-location lookup. <http://www.maxmind.com>.
- [6] Microsoft office 365. <http://office365.microsoft.com>.
- [7] Url encoding attack. <http://www.technicalinfo.net/papers/URLEmbeddedAttacks.html>.
- [8] Zoho online word processor. <http://writer.zoho.com>.
- [9] DINGLEDINE, R., MATHEWSON, N., AND SYVERSON, P. Tor: the second-generation onion router. In *Proceedings of the 13th conference on USENIX Security Symposium - Volume 13* (Berkeley, CA, USA, 2004), SSYM'04, USENIX Association, pp. 21–21.
- [10] HOWELL, J., JACKSON, C., WANG, H. J., AND FAN, X. Mashupos: Operating system abstractions for client mashups. In *Proceedings of the 11th USENIX workshop on Hot topics in operating systems* (Berkeley, CA, USA, 2007), HOTOS'07, USENIX Association, pp. 16:1–16:7.
- [11] PAVLO, A., AND SHI, N. Graffiti networks: A subversive, internet-scale file sharing model. *CoRR abs/1101.0350* (2011).
- [12] SHACHAM, H. The geometry of innocent flesh on the bone: Return-into-libc without function calls. In *Proceedings of CCS 2007* (2007), pp. 552–561.
- [13] WANG, H. J., FAN, X., HOWELL, J., AND JACKSON, C. Protection and communication abstractions for web browsers in mashupos. In *Proceedings of the 21st ACM Symposium on Operating Systems Principles* (2007).
- [14] WANG, H. J., GRIER, C., MOSHCHUK, A., KING, S. T., CHOUDHURY, P., AND VENTER, H. The multi-principal os construction of the gazelle web browser. In *Proceedings of the 18th Conference on USENIX security symposium* (2009).