OSS: Using Online Scanning Services for Censorship Circumvention

David Fifield[1], Gabi Nakibly[2], and Dan Boneh[1]

[1] Computer Science Department, Stanford University
[2] National EW Research & Simulation Center,
Rafael – Advanced Defense Systems Ltd.

Abstract. We introduce the concept of a web-based online scanning service, or OSS for short, and show that these OSSes can be covertly used as proxies in a censorship circumvention system. Such proxies are suitable both for short one-time rendezvous messages and bulk bidirectional data transport. We show that OSSes are widely available on the Internet and blocking all of them can be difficult and harmful. We measure the number of round trips and the amount of data that can be pushed through various OSSes and show that we can achieve throughputs of about 100 KB/sec. To demonstrate the effectiveness of our approach we built a system for censored users to communicate with blocked Tor relays using available OSS providers. We report on its design and performance.

1 Introduction

Nowadays many nations regularly filter Internet traffic by blocking news sites, social networking sites, search sites, and even public mail sites like Gmail. The OpenNet Initiative, which tracks public reports of Internet filtering, lists a large number of countries that filter Internet traffic. Over half of the 74 countries tested in 2011 imposed some degree of filtering on the Internet [1].

In response, several proxy systems have emerged to help censored users freely browse the Internet. Most notable among these is Tor [2], which, while originally designed to provide anonymity, has also seen wide use in circumvention. Other proposals include Telex [3], Infranet [4] and Ultrasurf [5] as well as several enhancements to Tor [6–8]. The existence of circumvention systems makes the censor's job harder: The censor must block all circumvention tools in order to remain effective.

Network censorship techniques fall into two broad classes: blocking by address and blocking by content. This work is mainly about the former: We seek to enable a censored user to communicate with a network host even when a censor blocks all traffic to and from that host's IP address. Flash proxy [6] is an example of a system resistant to address blocking; it creates a large number of short-lived proxies. Blocking by content, that is, the inspection of packet contents and other traffic characteristics such as timing, requires different circumvention techniques, for example mimicking other common protocols, as StegoTorus [7] does, or making the traffic look like no protocol in particular, as obfsproxy [8]

does. Combining resistance to both kinds of censorship is a subject of active development. Even though we are primarily concerned with blocking by address, Sect. 7 considers mitigations for content blocking in the system of this paper.

The system proposed in this paper is especially well suited to be used as a *rendezvous* protocol. A rendezvous protocol is an important component of a proxy-based circumvention system that allows a censored user to send a small amount of information (a few bytes) outside the censored region for the purpose of introducing the user to a proxy. Rendezvous protocols are low-bandwidth and designed to be difficult to block.

A complete circumvention system must also address secure client software distribution, an install system, and secure integration with a web browser. We have implemented our system as a Tor pluggable transport [9] so that it can use the Tor Project's existing infrastructure that addresses these concerns.

Our contributions. In this paper we propose a new approach to building proxies. In particular, we identify a large set of widely available web services that can be covertly made into proxies.

Our starting point is the observation that many web services take a URL as user input and then scan the web page behind that URL. We give many examples in Sect. 3, but for concreteness consider PDFmyURL, a service that does exactly what its name suggests: Given a URL it returns a PDF of the target page. With a URL as input, software on the server uses WebKit to fetch the page, render it, and convert it to PDF. The page is fetched immediately after the user clicks the submit button. We emphasize that pdfmyurl.com is just an example – there are *many* available services, including malware analysis sites and many others, that take a URL as input and then retrieve the page that the URL points to.

Now, suppose that the URL provided as input to pdfmyurl.com points to a site A.com that when accessed over HTTP returns a 302 redirect response to another site B.com. The server at pdfmyurl.com will dutifully follow the redirect and issue another HTTP request to the new target site B.com. Suppose now that B.com also returns a 302 redirect response back to A.com, but with a slightly modified path. pdfmyurl.com will follow the redirect back to A.com. Then A.com issues another redirect back to B.com and so on. This redirect ping-pong can go on for a while and the pdfmyurl.com server will obediently bounce back and forth between the two sites A.com and B.com. By embedding data in the URLs provided in each redirect, the two sites A.com and B.com can communicate using pdfmyurl.com as a proxy. Figure 1 illustrates this.

We refer to a service like PDFmyURL as an *Online Scanning Service* or OSS for short. For our purposes, an OSS must satisfy the following requirements:

1. It is not blocked by the censor.
2. It makes the initial HTTP scanning request in real time (within a few seconds of being asked).
3. It follows at least one redirect, where a "redirect" is any of a number of methods described in Sect. 4 (for example, we can use frames, refresh headers, or JavaScript to cause the redirects).

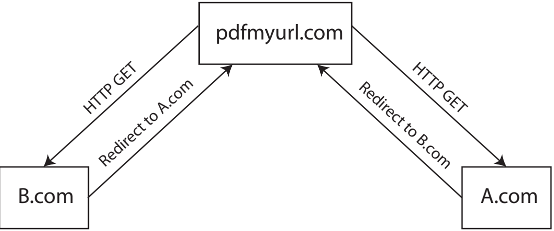

Fig. 1. Illustration of the redirection process.

Requirement 3 means that there is a way to respond to an OSS's HTTP request that causes the OSS to make another request to a URL of our choice. We use the OSS as a proxy by embedding data in these requests and redirecting it between two hosts. When used for rendezvous rather than bidirectional data transfer, Requirement 2 is relaxed (in some cases it is acceptable if rendezvous takes a few minutes) and Requirement 3 is unnecessary (rendezvous messages fit in a single request and do not require a reply).

There are many OSS-like services on the web. It is an advantage of this circumvention technique that there is no canonical list of "supported OSSes" that must be known in advance. One host may use any convenient OSS to communicate with the other, as long as it satisfies the three requirements above, without prior arrangement with the other host. What the OSS may do behind the scenes is not important for the purpose of data transfer; we use only its ability to make requests and follow redirects.

A comparison with the flash proxy system [6] is instructive. Where the flash proxy system uses the abundant resource of web browser IP addresses, our system uses online scanning services, which are less numerous but potentially more costly to block as they may host important services.

In principle, censors can counter this circumvention system by blocking all OSS providers. However, as we show in Sect. 3, OSSes are so prevalent that it would be difficult to discover all of them. Furthermore, blocking all OSSes on the Internet would cause economic hardship and block legitimate popular services that have nothing to do with censorship circumvention.

In Sect. 2 we detail the threat model we address in this paper. In Sect. 3 we survey many existing OSS services and survey which are suitable for circumvention and which have undesirable side effects. Section 4 describes a number of redirect methods and measure their performance with each OSS. Section 5 has measurements of the performance characteristics of several OSSes, with experimental results for overall throughput. In Sect. 6 we describe the system we built that allows censored users to communicate with blocked sites through arbitrary OSS providers. A security analysis of the system follows in Sect. 7. Ethical considerations of using OSSes in this way are the subject of Sect. 8. Section 9 concludes.

2 Threat Model

Our work deals with five entities, whose relationship is summarized in Fig. 2.

1. Censored user – a user within the filtered region who tries to access a target web site outside the filtered region.
2. Censor – an authority that monitors and blocks traffic between the censored user and the outside world. An example of a censor is a national government, censoring at the borders of a country. The censor is the adversary we try to circumvent.
3. OSS – an online web service used as an intermediary in communication. It is located outside the filtered region. It is assumed that the censor does not block traffic between the censored user and the OSS. The OSS may be oblivious to the circumvention effort: It does not generally actively assist circumvention, but neither does it work with the censor to frustrate circumvention.
4. Cooperating proxy – a server located outside the censored region that relays traffic to and from the target web site. An example of such a proxy is a Tor bridge. The censor blocks traffic between the censored user and the cooperating proxy; otherwise, the user would contact it directly.
5. Target web site – a web site located outside the filtered region. The censor blocks traffic between the censored user and the target web site.

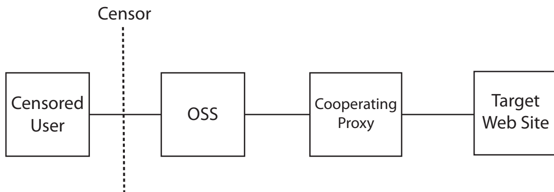

Fig. 2. Illustration of the relationship between the principal entities. Our system enables indirect communication between the censored user and the cooperating proxy, otherwise prohibited by the censor.

The censor may inspect all traffic passing through it and may block any packet it wishes. However, it does not do stateful tracking, namely keeping state on every monitored data session or IP endpoint. We assume the censor does not control the user's computer (through a backdoor or similar) and that the user is able to get and install circumvention software. The censor is motivated to minimize "collateral damage" caused by its blocking: Access to innocuous or economically important targets is allowed while a relatively small subset of traffic is blocked. In other words, the censor will not block web services that can be used as OSSes unless they are associated with circumvention – and even then, only if the cost of blocking them is not too high.

The OSS is independent of the censor and does not collude with the censor to prevent circumvention. The OSS is untrusted and may read or modify traffic passing through it.

Our security goals are to communicate with a blocked endpoint without exchanging traffic directly with it; to be expensive for the censor to block (in economic and social terms); and to be covert in the sense that it should not be possible to pinpoint a user without specifically focusing on that user's traffic.

3 Online Scanning Services

To demonstrate the feasibility of our circumvention method, we investigated a number of existing OSS providers listed below. We divide them into the categories of security, advertising, web diagnostics, processed retrieval, and link shorteners. In this section we list example OSSes in each category, and in the next we analyze their characteristics as applied to circumvention.

3.1 Security Scanners

This category consists of services that scan a web page for malicious content. The scans are initiated by submitting a URL in a web form.

Dr.Web `http://vms.drweb.com/online/`
An online scanner using the Dr.Web antivirus engine.

NoVirusThanks `http://vscan.novirusthanks.org/`
NoVirusThanks is a security company with a free multi-engine antivirus scan.

VirusTotal `https://www.virustotal.com/#url`
VirusTotal scans uploaded files and URLs with a variety of antivirus engines.

3.2 Contextual Advertising

Contextual advertising attempts to match advertisements to the web pages on which they appear. The ad network scans web pages containing advertisements to find keywords or other context.

Google AdSense `https://www.google.com/adsense/`
An AdSense advertisement is a piece of JavaScript code. It sends the URL of the page it is on to the AdSense servers in order to find out what to display. If the URL is not in the servers' cache, the servers begin an immediate scan. (The structure of our URLs ensures that they will not be found in cache.) Our experiments show that the AdSense crawler will crawl arbitrary URLs, not only those on a domain belonging to an AdSense customer.

3.3 Web Diagnostics

This category includes services that analyze the contents of a web page.

vURL Online `http://vurldissect.co.uk/`
Dissects web markup and extracts some information for analysis. The information includes such things as image and link references.

W3C Markup Validation `http://validator.w3.org/`
Checks the markup of web documents in formats like HTML and XHTML.

3.4 Processed Retrieval Services

Services that return a web document after filtering it in some way.

GoMo `http://www.howtogomo.com/`
Renders a web site as it appears on a smartphone.

IE NetRenderer `http://netrenderer.com/`
Renders a web site as it appears in various versions of Internet Explorer.

PDFmyURL `http://pdfmyurl.com/`
Converts a web page into PDF.

3.5 Link Shorteners

Link shorteners turn a URL and into another, usually shorter, URL that redirects to the original.

Google URL Shortener `https://goo.gl/`
The goo.gl shortener makes an HTTP request to the target page in order to show a thumbnail preview. We were not able to drive this OSS programmatically, as shortening a link and retrieving the preview is a complicated process involving a Google Account and JavaScript code. For this reason, and because initiating a scan has the side effect of creating a permanent short link, goo.gl is not a high-quality OSS for our purposes.

Twitter Link Shortener `https://t.co/`
Links posted on Twitter are shortened by the mandatory URL shortening service t.co [10]. As a side effect of creating a short link, a number of scanners retrieve the URL. To initiate a scan, we programmatically post a tweet containing the URL to be scanned. Like goo.gl, Twitter leaves a record of each scan request in the form of a short link and a tweet.

3.6 Discussion

The throughput of an OSS depends on the specific redirect method used, but overall some are faster than others. In addition, some OSSes are not suitable for circumvention: Twitter and goo.gl create a record of each communication in

the form of a short URL, and require using an account that ties circumvention to a long-term (potentially pseudonymous) identity. We make this division of tested OSSes into those that are fast, those that are slow, and those that have deficiencies that make them unsuitable for circumvention:

High-rate: Dr.Web, GoMo, NoVirusThanks, PDFmyURL.
Low-rate: AdSense, NetRenderer, VirusTotal, vURL, W3C.
Unsuitable: goo.gl, Twitter.

The fast OSSes can be used for bulk data transfer while slow ones are best suited for rendezvous. Section 5 tests a selection of the best OSSes and redirect methods.

The number of OSSes we study in this paper was limited only by our resources. Other potential OSSes are translation services, photo printing services, file hosts, RSS aggregators, and image sharing sites. We tested common web browsers as if they were OSSes, and found that browsers are capable of acting as circumvention proxies. However, proxying over redirects in browsers offers no real advantages over using custom code as in the flash proxy system [6], so we omit the results of browser testing.

4 Using an OSS for Communication

In this section we show how to use an OSS as a traffic relays. We then measure the performance characteristics of each of the OSSes from Sect. 3. In what follows we refer to the censored user as the *client*, and the remote cooperating proxy as the *server*.

Initiating communication. Each OSS is started differently. To name two examples, the W3C markup validator requires only a single GET request containing the URL to be scanned as a query parameter; VirusTotal requires first retrieving the home page to get a cross-site request forgery (CSRF) token, then including the token in a POST request along with the URL to be scanned. Once initiated, however, the client and server may communicate without knowing the details of the underlying OSS.

We describe a variety of techniques for causing an OSS to request a URL: HTTP redirects, refresh, frames, and JavaScript. We group all of these techniques under the general term "redirect methods." What they have in common is that they allow an HTTP response to control some aspect of a subsequent request. The easiest part of a request to control is the URL. Some redirect methods, such as the JavaScript-based ones, also allow control over the request body, which we take advantage of in order to send more data per round trip.

Relaying traffic. The URL is our primary vehicle for communication. Our URLs follow this format:

http://*host*:*port*/*random*/*id*/*seq*/*ack*?*query*

random is a random string, changing with every request, whose purpose is to inhibit caching by the OSS. *id* identifies the URL as being part of a particular data stream or session; it allows a server to handle multiple simultaneous clients. *seq* and *ack* carry information about what what bytes each endpoint has received. (OSSes do not in general follow an unlimited number of redirects; they effectively drop the last communication in a redirect chain. The client and server retransmit unacknowledged bytes until they are received.) *query* contains a `data` parameter encoding the payload (except for the redirect methods that carry data in the request body instead). `data` is base64-encoded using the URL-safe alphabet [11]. *query* additionally contains metadata like the client's "return address" (the URL to which the server's redirects will be directed), the name of the redirect method to use, and a `fin` marker for end-of-stream. A typical URL is

http://*host*:*port*/4a931e1d/e16813d8/0/0?data=SGVsbG8g...

Measurements. What follows in the rest of this section are descriptions of each redirect method, and the results of testing each redirect method against each OSS. In the performance tables, a redirect method/OSS combination is summarized by a triple of numbers. An example summary is

redirects	capacity	delay
10	2047	0.5

The three numbers in order are

– Maximum number of redirects followed. We cut off testing above 250 requests; services that did not stop before this limit are reported as ∞. The number of redirects matters because the client must kick off a new redirect chain once a previous chain is exhausted.
– Maximum data capacity per redirect, in bytes. For redirect methods that embed data in URLs, this is the maximum URL length allowed. The amount of useful payload data can be derived from the URL length by subtracting a constant to account for non-data parts of the URL, then multiplying by 3/4 to account for base64 encoding. For redirect methods that send data in request bodies, the number is the maximum body size allowed. We cut off testing above 512 KB; services that did not stop before this limit are reported as ∞.
– Delay, in seconds, between initiating a scan and when the OSS's first request is received. For some services the delay is variable, so we report an approximate average. For goo.gl, which was tested by manually copying URLs, the delay column is empty.

We tested the limits on number of redirects using a program that redirected to itself and kept a count of requests until a timeout. To measure payload limits, we did a binary search over URL and request body lengths.

Unusual results in the tables are called out with footnotes. We speculate that some services have limits in place other than those we measured, for example a limit on clock time. An unanticipated finding was that some services have

multiple backend scanners with different characteristics. For this reason, some table entries contain more than one number, separated by slashes. In the code samples that follow, `http://example.com/` takes the place of a data-carrying transport URL.

4.1 HTTP Redirects

The specification of HTTP 1.1 [12, Sect. 10] defines a number of status codes that effect redirects: 300 Multiple Choices, 301 Moved Permanently, 302 Found, 303 See Other, and 307 Temporary Redirect.

All these status codes have in common the Location header, whose value is the URL to redirect to. They have some differences, both in specification and in implementation. For example, 303 requires that the redirect be followed using the GET method, while 307 repeats the method used in the original request.

Table 1. Performance characteristics of HTTP redirects.

	300			301/302/303/307		
OSS	redirects	capacity	delay	redirects	capacity	delay
AdSense	0	0	0.3	5	2047	1.1
Dr.Web	∞	8181	0.5	∞	8181	0.5
GoMo	15	∞	3.5	15	∞	3.9
goo.gl	15	2047	–	15	2047	–
NetRenderer	10	2083	1.2	10	2083	1.2
NoVirusThanks[1]	10	$\approx$128000	1.3	10	$\approx$128000	1.4
PDFmyURL	0	0	0.9	∞	∞	1.1
Twitter[2]	0	0	2.5	4/25	>2047	2.3
VirusTotal[3]	5/20	2047	5.9	5/20	2047	4.2
vURL[4]	20	$\approx$128000	22.6	20	$\approx$128000	22.5
W3C	0	0	0.8	7	8181	0.7

[1] NoVirusThanks appears to be time-limited. We were able to send about 128000 bytes per redirect but the exact number fluctuated around this value.
[2] Posting on Twitter was observed to cause up to three requests from different /24 IP address ranges, each with its own characteristics. At least one of the scanners supports payloads of at least 2048 bytes.
[3] The behavior of VirusTotal was variable, probably because of different backend scanners. Sometimes 5 and sometimes 20 redirects were allowed. Sometimes the payload was limited to 2047 and sometimes apparently unlimited.
[4] vURL, like NoVirusThanks, appears to be time-limited.

Table 1 shows the performance of each OSS using 300-series redirects. We tested each of the five redirect codes separately. 301, 302, 303, and 307 had the same performance and level of support in our tests, so they are shown in a single table column. 300 was somewhat less widely supported. Support for HTTP redirects other than 300 is universal. HTTP redirects are the only working method for NoVirusThanks, Twitter, vURL, and W3C.

We generally embed data in URL parameters over GET requests. It is tempting to try to use 307 with a POST request in order to carry data in request

bodies, but unfortunately this does not work. After receiving a 307 redirect, the OSS makes a request to the new location using its original request body, not the body returned along with the 307. Changes to request bodies do not survive the "ping-pong" the way that changes to URLs do. In order to make use of POST and request bodies we turned to JavaScript-based redirects, described later.

4.2 Refresh

Another method of redirecting an HTTP request is the "meta-refresh" technique and the related Refresh header, which are widely supported despite being deprecated and non-standard [13]. A meta-refresh is a piece of HTML that instructs the user agent to go to another URL after a delay of 0 seconds:

```
<meta http-equiv="refresh" content="0; url='http://example.com/'">
```

The same effect can be accomplished with a Refresh HTTP header.

```
Refresh: 0; url='http://example.com/'
```

Table 2. Performance characteristics of HTTP refresh.

	meta-refresh			Refresh header		
OSS	redirects	capacity	delay	redirects	capacity	delay
AdSense	5	2047	1.3	5	2047	1.2
Dr.Web	0	0	0.6	0	0	0.4
GoMo	∞	∞	5.0	∞	∞	4.0
goo.gl	30	2047	–	30	2047	–
NetRenderer	1	2083	1.0	1	2083	1.0
NoVirusThanks	0	0	0.5	0	0	0.5
PDFmyURL	∞	∞	1.8	∞	∞	2.9
Twitter	0	0	1.0	0	0	2.6
VirusTotal[1]	0/≈150	0/∞	6.0	0/≈150	0/∞	4.5
vURL	0	0	22.3	0	0	22.9
W3C	0	0	0.8	0	0	0.8

[1] One of VirusTotal's scanners is apparently time-based; we were able to make about 150 refreshes in 60 seconds. Another scanner allowed no refreshes.

Meta-refresh and the Refresh header have identical performance characteristics within every OSS. Table 2 shows the performance of refresh-based redirects.

4.3 Frames

Recursively loaded resources can serve the same purpose as redirects. An HTML document may contain a frameset or iframe; those may in turn contain frames, and so on, up to an implementation-defined limit on nesting. Using the frameset method, each response contains HTML like this example:

```
<frameset><frame src="http://example.com/"></frame></frameset>
```

Using iframe, responses look like this:

```
<iframe src="http://example.com/"></iframe>
```

Table 3 shows how frames are treated by each OSS. Limits on frame nesting vary. PDFmyURL, which does not limit other redirect methods, limits frame nesting to 201 levels.

Table 3. Performance characteristics of HTML frames.

	frameset			iframe		
OSS	redirects	capacity	delay	redirects	capacity	delay
AdSense	0	0	0.6	0	0	0.7
Dr.Web	1	≈450000	0.6	1	≈450000	0.5
GoMo	201	∞	3.8	201	∞	4.3
goo.gl	9	2047	–	9	2047	–
NetRenderer[1]	≈80	2083	1.6	≈80	2083	1.2
NoVirusThanks	0	0	0.6	0	0	0.5
PDFmyURL	201	∞	1.8	201	∞	1.7
Twitter	0	0	2.0	0	0	2.2
VirusTotal[2]	0/9	0/∞	5.0	0/9	0/∞	9.3
vURL	0	0	22.4	0	0	22.3
W3C	0	0	0.6	0	0	0.7

[1] NetRenderer's limit appears to be time-based. We were able to make about 80 redirects in 30 seconds.

4.4 JavaScript

JavaScript provides another way for one web resource to load another. JavaScript has the ability to control request bodies via the POST method – an advantage because services typically allow more data in request bodies than in URLs.

JavaScript-based redirects' higher bandwidth is offset by less widespread support. Only goo.gl, GoMo, NetRenderer, and PDFmyURL usefully followed JavaScript-based redirects.

We tried two different ways of loading a URL in JavaScript. The first builds an HTML form containing the payload and submits it in the document's `onload` handler:

```
<body onload="document.f.submit();">
  <form name="f" method="post" action="http://example.com/">
    <input name="data" value="SGVsbG8g..."/>
  </form>
</body>
```

The second uses XMLHttpRequest to load a new HTML page (which contains its own XMLHttpRequest), and replaces the current page with the loaded page.

```
<script type="text/javascript">
  xhr = new XMLHttpRequest();
  xhr.open("POST", "http://example.com/");
  xhr.setRequestHeader("Content-type", "application/x-www-form-urlencoded");
  xhr.onreadystatechange = function() {
    if (xhr.readyState == xhr.DONE && xhr.status == 200) {
      document.open();
      document.write(xhr.responseText);
      document.close();
    }
  };
  xhr.send("data=SGVsbG8g...");
</script>
```

Table 4 is a summary of JavaScript-based redirects. goo.gl, which tightly limits the lengths of URLs to 2047 bytes, has no equivalent restriction on the length of request bodies.

Table 4. Performance characteristics of JavaScript redirects.

	onload			XMLHttpRequest		
OSS	redirects	capacity	delay	redirects	capacity	delay
AdSense	0	0	0.5	0	0	0.7
Dr.Web	0	0	0.6	0	0	0.5
GoMo	∞	∞	7.1	∞	∞	3.2
goo.gl	30	∞	–	15	∞	–
NetRenderer	0	0	1.0	0	0	1.0
NoVirusThanks	0	0	0.5	0	0	0.7
PDFmyURL	∞	∞	2.5	∞	∞	1.4
Twitter	0	0	2.6	0	0	2.9
VirusTotal	0	0	4.4	0	0	4.0
vURL	0	0	22.3	0	0	22.5
W3C	0	0	0.7	0	0	0.7

5 Experiments

5.1 Implementation

To evaluate the performance of the circumvention scheme we implemented client and server programs. Both programs have similar capabilities: Each listens for HTTP requests, identifies the stream to which an incoming request belongs, extracts the payload, and redirects the request back to the peer with a portion

of buffered payload data. The client program presents a SOCKS [14] interface to the local host. The programs can be used with or without Tor, and we tested both configurations.

The main difference between the programs is that only the client initiates scan requests to OSSes, while the server remains passive and must wait for an HTTP requests from an OSS before sending data back to the client. This design choice requires the client to refresh redirect chains when they reach their limits, and to poll the server for data periodically, even if the client has nothing to send. This model has advantages but is not the only possibility; see Sect. 6 for further discussion.

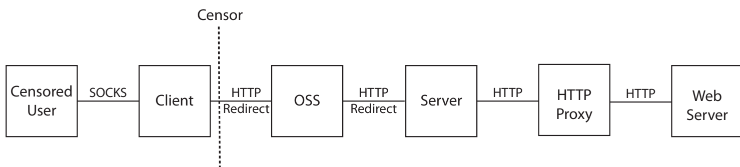

Fig. 3. End-to-end data flow between a censored user and an HTTP proxy.

Figure 3 shows an end-to-end flow for communicating with an HTTP proxy. The censored user connects to the client with SOCKS and requests a connection to the server. The client program chooses an OSS and initiates a request for it to scan the server. The scan request contains a random stream ID, a return address on which the client is listening, and a redirect method. The redirect method is needed because the server may not know what redirect methods the OSS supports. The server receives the scan request, extracting the stream ID and data payload. The server immediately redirects the OSS back to the client's return address. The server is configured to feed its concatenated payloads to the HTTP proxy. When the HTTP proxy returns data to the server, the server buffers it until it receives another request from the client with the same stream ID to which it can respond with a payload.

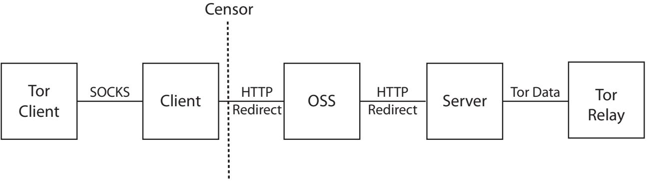

Fig. 4. End-to-end data flow between a censored user and a Tor relay.

Figure 4 shows the end-to-end flow when configured to use Tor. The use of a SOCKS interface means that both client and server work as Tor pluggable transports [9]. In this configuration, the Tor relay serves as a general-purpose proxy. Not shown are the additional hops that packets would take through the Tor network before reaching the target web server.

5.2 Throughput Measurements

We now present the results for the throughput measurements for different OSS and redirection method combinations. For these measurements we used only a single redirect chain at a time. After a chain is exhausted as a result of reaching the redirection limit, a new scan request is initiated. In these tests, both client and server programs were on the same host, that is, traffic went out over the Internet and then returned to the same place. We timed the download of a 1 MB file. The available bandwidth between our test machine and the location of the remote file is high enough that the time taken to download mostly reflects OSS overhead. Table 5 lists the measured throughput for each OSS and redirect combination. In general, two different redirection methods in the same category yield roughly similar results given the same OSS. If there is a difference we show the higher throughput.

Table 5. Throughput measurement results (bytes per second).

OSS \ method	HTTP redirects	refresh	frames	JavaScript
AdSense	500	500	–	–
Dr.Web	20K	–	–	–
GoMo	22K	28.2K	28.2K	175K
goo.gl	350	400	410	110K
NetRenderer	850	–	1.3K	–
NoVirusThanks	21K	–	–	–
PDFmyURL	220K	160K	180K	265K
Twitter	2.4K	–	–	–
VirusTotal	1K	–	–	–
vURL	250	–	–	–
W3C	4.6K	–	–	–

Though Sect. 3 has examples of OSSes allowing half a megabyte or more of payload, larger payload sizes bring diminishing improvements in asymptotic bandwidth, while increasing latency. In our transport programs, we limited payloads to 32 KB.

It is evident from Table 5 that the throughputs of different OSS/redirect combinations vary widely. The lowest throughput we measured, 250 B/s, was with vURL/302 and the highest, 265 KB/s, came from PDFmyURL/onload.

The user may increase available throughput by simultaneously initiating multiple scan requests, to the same OSS or to different OSSes. We tested this technique for every OSS and redirection method by creating two concurrent streams, and observed their aggregate throughput to be twice that of a single stream, with the exception of AdSense. AdSense seems to impose a limit on the resources allocated at any given time to handle scan requests received from the same IP address. Two simultaneous scan requests caused the time for the OSS to act upon a received redirect to double. The other OSS that is operated by Google, goo.gl, does not impose such limits.

We repeated the tests using with the programs configured to use Tor. We tested several OSS and redirection method combinations and found that they generally produce about the same throughput as with a bare HTTP proxy.

6 The Overall System

Redirect techniques in OSSes can be used to carry data – in some configurations with high enough bandwidth and low enough latency to support comfortable web browsing. In this section we describe how to use this facility as part of a larger circumvention system in two scenarios: as a rendezvous method and as a bulk bidirectional data transport.

Anonymity is important for circumvention. For this reason, we use Tor as the target of OSS proxies, even though the idea of using HTTP requests for communication is not tied to Tor. Traffic passed over Tor is encrypted, so it is unreadable by the censor and, importantly, by the OSS. Tor traffic is additionally authenticated, so that a malicious OSS cannot cause communication to be redirected to something other than a Tor relay without the client noticing. Tor by itself is widely used for circumvention, but in places where Tor is blocked, it should not be necessary to give up on Tor's protections while using an OSS.

In the discussion so far, redirects have been used as a means of decreasing delay, with only the client making HTTP requests and initiating new redirect chains. Before continuing, we note an alternative design that has both client and server mutually initiating single scans of each other. This alternative has the advantage of not requiring the OSS to support redirects, relying only on its ability to make single requests. Disadvantages are higher overhead and delay – both client and server now receive the HTTP response to their every scan request, in addition to the HTTP requests made by the OSS – as well as increased server-side complexity. Another disadvantage is that the circumvention server has to support all of the potential OSSes, and know how to initiate a request with each of them. This eliminates the client's ability to choose its own unblocked OSS independently of the circumvention server.

An important property of the system is that every user has the freedom to choose which OSSes will be used without coordination with the cooperating proxy, the OSS, or other users. Even without such coordination, finding OSSes in the first place requires some expertise of the user. We leave unspecified how a user might find OSSes on an ongoing basis.

Both censored client and cooperating proxy act as HTTP servers that must be able to receive requests from the OSS, which acts as an HTTP client. This unfortunately requires that both client and server be able to receive TCP connections, which in particular means that neither may be behind network address translation (NAT). The cooperating proxy is a server on the Internet that can be assumed to be able to receive connections, but many censored users in the real world are behind NAT. For those users, using an OSS for communication will require the technical ability to do port forwarding, when port forwarding is even possible. NAT does not pose a problem for rendezvous, which only requires sending in one direction.

6.1 OSS as Rendezvous

Recall that a rendezvous protocol allows a censored client to send a small amount of information out through the censor in a way that is very hard to block. The usual goal is to bootstrap a higher-bandwidth transfer mechanism. Rendezvous over OSS is simple and doesn't even involve redirects. The client just encodes the data it wants to send in a URL pointing to the server it wants to send the data to. Rendezvous messages are short, so one request is enough, and the client doesn't have to receive a reply. If the OSS uses encryption, it is not possible for the censor to selectively block rendezvous messages by looking for distinctive URLs. It may be expensive to block all traffic to an OSS, if the OSS is commonly used for purposes other than circumvention.

Rendezvous is by nature low-bandwidth and infrequent. Ideally, just one short message is sent at the beginning of a session lasting hours. There is a lower risk of OSS administrator annoyance and blocking when the system is used in this way, compared to high-bandwidth uses.

There exists the danger of an eavesdropping OSS; the OSS should be regarded as an untrusted network router capable of seeing the traffic that passes through it. Our implementation of OSS-based rendezvous for the flash proxy system additionally encrypts messages before encoding them into URLs, as a measure of protection against OSSes that log the requests they are asked to make.

6.2 OSS as Bidirectional Data Channel

It certainly works to use a public OSS connected to a Tor relay as a general-purpose proxy. With heavy use, though, there is a chance of detection and blocking, not by the censor but by the OSS itself. (During our tests, we were blocked by PDFmyURL and Twitter.) The operators of an OSS cannot be assumed to care about the cause of circumvention, and a large number of unusual requests and redirects are likely to be unwanted if noticed. As an example, PDFmyURL's page on overusage [15] explicitly lists the conditions that may result in blocking:

> *Your IP or domain can be excluded if you overuse the service. You'll get a blocked message if you make a combination of:*
>
> 1. *more than 100 PDFs in two hours with a single IP, and*

2. *all these 100+ PDFs take more than 1000 seconds to process on our servers, and*
3. *use more than 10% of CPU resources.*

We suspect that what led to our blocking was heavy bandwidth tests, which transferred several megabytes using URLs as long as 32 kilobytes. Shorter and less bandwidth-intensive applications are less likely to be problematic. While using an OSS as a data channel may be useful in some special circumstances, we think that use as a rendezvous is more universally applicable.

7 Security Analysis

In this section we evaluate the system for unique traffic characteristics. We analyze each characteristic relative to the threat model outlined in Sect. 2 and discuss whether it can be used by the censor to flag traffic produced by our mechanism. In addition, we consider some possible extensions to the system that increase its resistance to content blocking.

Cooperating proxy address. Both the initial scanning requests as well as the redirection responses coming from the censored user contain a URL, the IP address or domain name of which belongs to the cooperating proxy. The censor can block flows that contain the address or domain name of a blocked host.

It may seem that HTTPS encryption is the easy solution to this problem, but the censored client can't be assumed to have a certificate that will be considered valid by the OSS. Requests to the client, and the client's redirection responses, must be in plain HTTP. Using the alternative design mentioned in Sect. 6 that does not use redirects, HTTPS works to hide the cooperating proxy's address; as there are no redirects, there is no need to send the location to redirect to.

A mitigation of the problem is to obfuscate the proxy's name in the traffic that goes between the censored user and the OSS. There are several ways to do this obfuscation. With the JavaScript redirection methods, the censored user can use code to generate the proxy address in URLs, rather than embedding the address directly. Blocking becomes more costly as the censor must run JavaScript code to find the other endpoint of a communication.

Domain names are a cheap and abundant resource. Multiple domain names that resolve to the same IP address serve as aliases that increase the cost of blocking. Using a rendezvous protocol, every censored user may learn a single domain name alias for their chosen OSS.

Instead of using many domain names for the proxy one can use many shortened URLs that all point to the same proxy name. To do this one must use a URL shortening service that allows to attach to the shortened URL custom parameters. An example shortener that allows parameters is `http://para.ms/`. It copies parameters attached to the shortened URL to the original long URL. This way the censored user may request that the OSS scan the shortened URL with custom parameters and the OSS will be redirected to the proxy using the same parameters.

Incoming connections. A censored user receives incoming connections from the OSS. By disallowing incoming connections to residential users a censor would be able to block our mechanism. Although it is much more common for a residential user to initiate connections rather than receive them, it is our assumption that incoming connections are common enough that the censor will be reluctant to block them wholesale.

An alternative blocking strategy is be to block only incoming requests coming from known OSSes. This strategy will have less collateral damage. However, as we have shown potential OSSes are abundant and it is hard for the censor to discover and block them. Furthermore, blocking incoming connections from known OSSes may incur economic damage. For example, disallowing Google from initiating connections to the filtered region would inhibit web sites inside the filtered region from appearing on Google's search results or from using AdSense.

URL pattern. The system uses distinctive URLs containing hexadecimal strings and well-known query string parameters. An alternative URL design may encode all the information in a single query string parameter with a predefined obfuscated structure. The parameter name may be arbitrary. It may be necessary to avoid using very long URLs (the censor may filter on length), potentially reducing bandwidth.

Number of redirects. A censored user responds to HTTP requests with some form of redirection. It may be suspicious if a large fraction of HTTP responses are redirects. However, only HTTP redirects and refresh redirects are straightforward to detect. In addition, a censor limited as described in our threat model (Sect. 2) is not able to do stateful monitoring on the entire volume of monitored traffic. Therefore, it is impossible for the censor to gather the number of redirects for every endpoint in the filtered region. The censor might be able to do stateful monitoring only of connections to known OSSes. However, as we have stated above it is hard for the censor to discover all potential OSSes.

Number of outgoing connections. A censored user may initiate many outgoing connections to the same OSS. An large number of outgoing connections may be suspicious. Again, due to the stateless monitoring constraint the censor is not be able to measure the number of outgoing connections for every endpoint in the filtered region. Statefully monitoring only those endpoints that communicate with known OSSes may be feasible, but it is hard for a censor to discover all potential OSSes.

Eavesdropping by the OSS. Our threat model assumes that the OSS is not colluding with the censor; however it is not necessarily a trusted entity. An OSS can intercept all communications between the user and the cooperating proxy. In this sense the OSS resembles an ISP, Tor entry relay, or other network router that lies on the path between the user and the cooperating proxy. Traffic should be encrypted and authenticated, as the Tor protocol is, to prevent eavesdropping and tampering by the OSS.

8 Ethical Considerations

Our proposal uses web services for a purpose they were not designed for. Fortunately, there are measures that will allow an OSS to curb or eliminate its use as a circumvention relay.

One such measure is for the cooperating proxy (i.e. the Tor bridge) to send with every redirect message a signal that will indicate that this redirect is part of a censorship circumvention data flow. If an OSS does not wish to take part in the data flow, it may refuse to follow that redirect. The signal between the cooperating proxy and the OSS can take many forms as long as it is known to all proxies and OSSes. An example of such a signal is a distinguished HTTP header included in the HTTP response to the OSS. Note that this signal does not compromise the security of the circumvention method because the traffic between the proxy and the OSS is not monitored by the censor.

9 Conclusions

We have shown that general-purpose web services that scan a given URL can be covertly used to relay information between a user in a censored region and a blocked web site. Blocking *all* these general-purpose services is practically impossible to do and would severely cripple the web in the censored region, disabling security scanners, URL shortening services, advertisers, and many others.

After identifying a large class of online scanning services (OSSes), we analyzed their performance as proxies. We showed that many common services can handle many round trips and provide decent throughput. Some OSSes can be used to proxy live session data while others provide limited bandwidth and would mostly be used only for rendezvous. We experimented with a system that can relay information between a browser and server using these OSSes as a relay. The system is available as an experimental rendezvous for the flash proxy system [6] and is part of Tor's pluggable-transports web browser bundles starting with the 2.4.11-alpha-1 release [16]. Source code and experimental results are available from `https://gitweb.torproject.org/user/dcf/oss.git`.

Acknowledgments

The work is supported by DARPA and Space and Naval Warfare Systems Center Pacific under Contract No. N66001-11-C-4022. Opinions, findings and conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect the views of the Defense Advanced Research Project Agency and Space and Naval Warfare Systems Center Pacific. Distrib. Statement "A:" Approved for Public Release, Distribution Unlimited.

References

1. The OpenNet Initiative: OpenNet Initiative Internet censorship data. `http://opennet.net/research/data` (November 2011)

2. Dingledine, R., Mathewson, N., Syverson, P.: Tor: The second-generation onion router. In: Proceedings of the 13th USENIX Security Symposium. (August 2004)
3. Wustrow, E., Wolchok, S., Goldberg, I., Halderman, J.A.: Telex: Anticensorship in the network infrastructure. In: Proc. 20th USENIX Security Symposium. (2011)
4. Feamster, N., Balazinska, M., Harfst, G., Balakrishnan, H., Karger, D.: Infranet: Circumventing web censorship and surveillance. In: Proceedings of the 11th USENIX Security Symposium. (2002)
5. Ultrareach Internet Corp.: Ultrasurf proxy. `http://www.ultrasurf.us/`
6. Fifield, D., Hardison, N., Ellithorpe, J., Stark, E., Boneh, D., Dingledine, R., Porras, P.: Evading censorship with browser-based proxies. In: Proceedings of PETS 2012. Number 7384 in LNCS (2012) 239–258
7. Weinberg, Z., Wang, J., Yegneswaran, V., Briesemeister, L., Cheung, S., Wang, F., Boneh, D.: StegoTorus: a camouflage proxy for the Tor anonymity system. In: ACM Conference on Computer and Communications Security. (2012) 109–120
8. Kadianakis, G., Mathewson, N.: Obfsproxy architecture. `https://www.torproject.org/projects/obfsproxy` (December 2011)
9. Appelbaum, J., Mathewson, N.: Pluggable transports for circumvention. `https://gitweb.torproject.org/torspec.git/blob/HEAD:/proposals/180-pluggable-transport.txt` (October 2010)
10. Twitter: FAQs about Twitter's link service. `https://support.twitter.com/entries/109623`
11. Josefsson, S.: The base16, base32, and base64 data encodings. RFC 4648 (Proposed Standard) (October 2006)
12. Fielding, R., Gettys, J., Mogul, J., Frystyk, H., Masinter, L., Leach, P., Berners-Lee, T.: Hypertext transfer protocol – HTTP/1.1 (1999)
13. Jacobs, I., Chisholm, W., Vanderheiden, G.: HTML techniques for web content accessibility guidelines 1.0. Technical report, W3C (December 2000) `http://www.w3.org/TR/2000/NOTE-WCAG10-HTML-TECHS-20001106`. Latest version available at `http://www.w3.org/TR/WCAG10-HTML-TECHS/`.
14. Leech, M., et. al.: SOCKS protocol version 5 (1996)
15. PDFmyURL: Over usage (limited use). `http://support.pdfmyurl.com/topic/getting-help-overusage` (October 2011)
16. Fifield, D., Allaire, A.: Ticket #7559: Registration via indirect URL request. `https://trac.torproject.org/projects/tor/ticket/7559` (March 2013)