Uses and Abuses of Server-Side Requests

Giancarlo Pellegrino(✉)[1], Onur Catakoglu[2],
Davide Balzarotti[2], and Christian Rossow[1]

[1] CISPA, Saarland University, Saarland Informatics Campus
{gpellegrino,crossow}@cispa.saarland
[2] Eurecom
{onur.catakoglu,davide.balzarotti}@eurecom.fr

Abstract. More and more web applications rely on server-side requests (SSRs) to fetch resources (such as images or even entire webpages) from user-provided URLs. As for many other web-related technologies, developers were very quick to adopt SSRs, even before their consequences for security were fully understood. In fact, while SSRs are simple to add from an engineering point of view, in this paper we show that—if not properly implemented—this technology can have several subtle consequences for security, posing severe threats to service providers, their users, and the Internet community as a whole.
To shed some light on the risks of this communication pattern, we present the first extensive study of the security implication of SSRs. We propose a classification and four new attack scenarios that describe different ways in which SSRs can be abused to perform malicious activities. We then present an automated scanner we developed to probe web applications to identify possible SSR misuses. Using our tool, we tested 68 popular web applications and find that the majority can be abused to perform malicious activities, ranging from server-side code execution to amplification DoS attacks. Finally, we distill our findings into eight pitfalls and mitigations to help developers to implement SSRs in a more secure way.

1 Introduction

Web applications have evolved from purely client-to-server patterns to an intertwined network of multiple web services. As a consequence, an increasing number of web applications retrieve external resources provided by other web services, often steered by user inputs. For example, social networks regularly fetch pages to display image and video previews of links posted by users, online calendars can import remote iCal data, web mail clients fetch emails from user-provided inboxes, and online image editors retrieve images from user-provided URLs. Such service-to-service communication is also integrated into business web applications and is at the core of several web-based protocols (e.g., OpenID and SAML) and Cashier-as-a-Service web applications (e.g., online stores using PayPal Express Checkout).

To support service-to-service communication, web applications rely on *server-side requests* (SSRs), which are HTTP requests generated by a server towards

another web service. SSRs are often used to avoid passing relay messages between different services via the user, or to allow complex services to perform requests outside the boundaries of the same origin policy. Unfortunately, although the communication between web services is not new, we noticed an alarming lack of information and understanding regarding the threats and the security implications of this communication pattern. For example, when a user posts a URL to a social network, the server-side web application automatically fetches the content from the URL to display a visual preview of the page. However, giving the user the freedom to choose the URL means that she can control the destination and potentially also the content of SSRs. This communication pattern is getting more and more common to improve user experience and provide advanced features in a wide range of applications. Unfortunately, as is often the case for emerging web technologies, developers are often too quick to jump on the bandwagon without fully understanding the risks for security. In fact, as we present in this paper, SSRs are difficult to get right and, if not properly implemented, they can be abused to conduct malicious actions against the service itself, its users, or even third-party web applications.

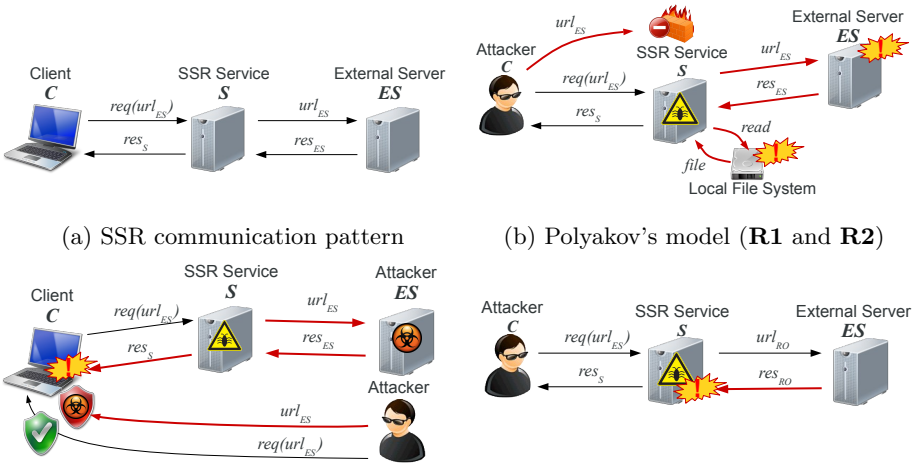
Existing work in this field focuses on *Server-Side Request Forgery* (SSRF), a family of software vulnerabilities that allow an attacker to misuse SSRs to perform port scans [27, 15] and buffer overflows [22]. However, this is only the tip of the iceberg of the possible security flaws that affect this communication pattern. Unfortunately, to date, we still lack a complete picture of the threats posed by SSRs.

To shed some light on the risks of this communication pattern, in this paper we present the first extensive assessment of the security implications of SSRs. We first present a classification to propose a common terminology for future research in the field. Our classification groups SSRs according to the level of control the attacker has, the role played by the vulnerable systems, and the potential attack targets. We then apply our classification to introduce four attack scenarios in which seemingly innocuous services can be composed together to form sophisticated attacks. For example, we show how popular services can be abused to distribute links to phishing pages—bypassing existing URL blacklists and reputation services.

In order to understand how widespread the problem is and what the most common mistakes are, we propose a tool called `GÜNTHER` and use it to analyze 68 web applications that accept user-provided URLs. We found weaknesses and security risks in 52 of them. Finally, to help developers to take more informed decisions and reduce the risks associated with this delicate communication pattern, we distilled our findings in a list of eight security-related recommendations.

To summarize, this paper makes the following contributions:

- We propose a new classification to classify SSRs;
- We present four new attack scenarios in which SSRs can be used to mount sophisticated Denial-of-Service (DoS) attacks, deliver malware, and bypass client-side countermeasures. We show that SSRFs are only one of the possible security flaws introduced by SSRs.



(a) SSR communication pattern

(b) Polyakov's model (**R1** and **R2**)

(c) Attack against C (**R3**)

(d) Attack against S (**R4**)

Fig. 1: SSR pattern and risks

– We discuss the results of the experiments we conducted on 68 web services, 54 of which we found to be affected by at least one security flaw.
– We present a clear set of mitigations to help developers to implement SSRs in a more secure way.

2 Background

In this section, we present the SSR communication pattern, and we elaborate on its use in modern web applications. Then, we present an overview of the threat models, and finally, we present the current understanding of the security risks.

2.1 Server-Side Request Communication Pattern

The SSR pattern is shown in Figure 1a. It involves three entities: a client C, an SSR service S, and an external server ES. The protocol starts when C sends an HTTP request $req(url_{ES})$ to S containing a user-specified url_{ES}. The position of url_{ES} in the HTTP request is application-specific, e.g., it could be inserted in the query string, in the POST data, or even in the resource field of the HTTP request. S extracts the URL and initiates a connection to fetch the corresponding resource res_{ES} from the remote server ES. Depending on the use of SSR, S can forward the resource res_{ES} back to C (i.e., $res_S = res_{ES}$), or return the result of a transformation (i.e., $res_S = f(res_{ES})$). For instance, S can embed res_{ES} into res_S, or simply return an identifier of the retrieved resource.

Use Cases—SSRs are widely used in web applications. For example, social networks use SSRs to retrieve user-provided URLs and share them on the user's

page. SSRs are also common in business applications, such as web office suites, in which they are used to include online resources (e.g., pictures) in documents. The list also includes online development tools, news aggregators, and image processing applications. Online development tools help developers, for example, to validate documents such as XML or JSON objects, or allow developers to test their web application with different browsers. SSRs are also at the core of news aggregators, which retrieve news from newspapers or RSS documents. Another use of SSRs is in web-based security protocols, such as the OpenID authentication protocol [8]. In OpenID, a client wants to be authenticated at the service provider (SP) by using her own credentials at the identity provider (IdP). OpenID allows the two providers to communicate either indirectly, i.e., by using the client's browser as a relay agent, or directly via SSRs. In this case, the SP acts as an HTTP client and initiates the connection with the IdP, which in turn acts as an HTTP server. SSRs are also used in other web-based security protocols, such as SAML SSO.

Server- vs. Client-Side Requests—The counterparts of SSRs are client-side requests (CSRs) in which C retrieves a resource at ES and sends it to S. However, replacing SSRs with CSRs may not be practical, secure, or efficient.

Practicality: CSRs can be implemented with cross-origin requests (CORs) in which a resource in the domain of ES is transmitted to S. These requests are subject to the same-origin policy (SOP for CORs) and the cross-origin resource sharing mechanism [26] (CORS). The former forbids accessing resources in a domain (i.e., ES) of a different origin from the request (i.e., S). These requests can be relaxed with CORS; however, CORS assumes a pre-established agreement between two different domain origins to allow requests from one to access resources of the other. This solution is often not practical because each service needs to keep and maintain a whitelist of domains that can access their services, and developers may not be able to modify the whitelist of third-party services. This has spurred the development of techniques to circumvent these obstacles, e.g., to bypass SOP for CORs (often considered to be security flaws, such as JSONP), or using the more flexible SSR paradigm.

Security: In protocols like OpenID, the involved parties do not agree on shared secrets such as cryptographic keys. Instead, they generate or exchange keys during the protocol run. In contrast to SSRs, CSRs may expose keys to attackers, thus endangering the validity of the authentication process.

Efficiency: CSRs may introduce additional costs. For example, social networks and online tools for developers may need to retrieve several resources to create a synthesis of the web page or to analyze its content. For each resource, an SSR service will issue one request and one response. With CSRs, on the other hand, the number of messages can double: The first request-response pair retrieves the resource from ES, while the second pair uploads the resource to S for further processing.

2.2 Security Risks and Threat Models

While SSR is a useful communication pattern which enables service-to-service communication, if not properly implemented it can be abused to perform a wide range of malicious activities, such as:

R1 SSRs can be abused as stepping stones to attack ES, for instance by performing denial-of-service attacks against Internet-facing services. Other attacks can be against services of S's private network.

R2 S may accept untrusted URLs which reference local resources, e.g., files hosted by S. For example, this attack can be used to exfiltrate system configuration files, passwords, and databases.

R3 SSRs introduce a new level of indirection between web browsers and the origin of resources. As a result, browsers may no longer be able to determine the real origin of a page, thus leaving users exposed to malicious content such as malware.

R4 Vulnerabilities in S can be exploited with incoming responses from ES. Responses may be processed to generate res_S for C. An adversarial ES can potentially craft malicious messages res_{ES} with the purpose of exploiting vulnerabilities in S.

These risks are shown in Figure 1b (for **R1** and **R2**), Figure 1c (for **R3**), and Figure 1d (for **R4**). Figure 1b corresponds to the initial threat model proposed by Polyakov et al. [22]. The entities of Polyakov's model are an attacker C, an SSR service S, a service ES, the file system of S, and a firewall. C aims to access ES or the local file system of S. However, ES is protected by a firewall that blocks direct access from the Internet. S is exposed both to the Internet and to the local network. If not carefully implemented, an attacker can abuse SSRs performed by S to access internal servers that are in S's network, i.e., **R1**, or even retrieve files from S (e.g., via the `file://` protocol), i.e., **R2**.

Unfortunately, Polyakov's threat model is not complete as it neglects C as a possible victim (i.e., **R3**) and it considers only a fraction of the attack surface of S, thus ignoring other threats (i.e., **R4**). In this paper, we propose a more complete threat model that also incorporates new attacks in which SSRs are abused to target C (see Figure 1c) and S (see Figure 1d). In Figure 1c, ES hosts malicious content and C is an honest client that adopts URL-based countermeasures to protect the user from malicious content (such as filtering mechanisms like Google Safe Browsing). The attacker targets C by tricking the user into visiting the malicious page ES, possibly abusing an innocent but vulnerable S. While C may believe she is visiting a well-reputed service S, in fact, S may just act as a proxy for malicious content hosted at ES, effectively circumventing any reputation-based mechanisms deployed by C. In Figure 1d, the attacker is C, whereas ES hosts malicious content. The attacker submits the URL of the malicious content to S, which fetches res_{ES} and processes it. For example, if S implements poor resource validation mechanisms, it may be susceptible to resource exhaustion attacks via specially-crafted resources.

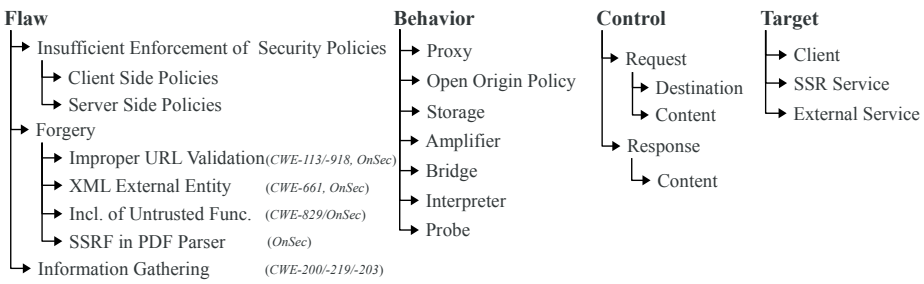


Fig. 2: SSRs classification

2.3 Awareness of the Security Risks

A closer look in the academic and non-academic literature and developer best practices (e.g., design patterns, coding rules, and API documentation) shows that (i) SSRs have received no attention by academic literature and (ii) existing non-academic works focus exclusively on Polyakov's threat model and thus neglect threats against C and S. In addition, despite the popularity of the attacks in Polyakov's threat model, there is a lack of documentation describing proper ways to implement SSR services and attack countermeasures. As a result, developers may develop vulnerable SSR services that can be abused by attackers.

3 SSR Classification

Despite anecdotal evidence, to date there is no systematic study of the SSR communication pattern. Therefore, we introduce a classification that proposes a common terminology for us and for future researchers. Our classification (Figure 2) includes and supersedes pre-existing categorizations, classifying SSRs according to four different directions: *flaws*, attacker *control*, S's *behavior*, and *victims*. To the best of our knowledge, this is the most extensive existing classification of SSRs.

The four dimensions of our classification are not mutually exclusive. In fact, services often play multiple roles and may suffer from multiple flaws. For this reason, our classification cannot be considered a *taxonomy*. Furthermore, our classification is based on the current knowledge of SSR abuse which may change. However, while target and control dimensions covers all possible combinations, flaw and behavior dimensions are an enumeration and thus may be incomplete. As the popularity of the SSR pattern increases, new types of vulnerabilities and behaviors can be detected. Nevertheless, new discoveries can be used to extend both flaws and behaviors dimensions. In the rest of this section, we describe each category in more detail.

3.1 Flaw-Based Classification

The first classification is based on the type of flaw of S. A flaw can occur when S accepts and processes inputs from C, and when S accepts and processes the resource res_{ES}. This classification includes known vulnerabilities, i.e., forgery and information disclosure vulnerabilities of the CWE database and OnSec classification. Additionally, we extend it with a new class of vulnerabilities called *insufficient security policy enforcement*.

Insufficient Security Policy Enforcement—An HTTP conversation between the browser of C and ES can involve different security policies. For example, C may use URL-based reputation lists to avoid visiting malicious pages. Similarly, the server may restrict access to its resources, e.g., by using the `Access-Control-Allow-Origin` header (ACAO, for short). The problems arise when S acts as an intermediary and it fails to enforce the aforementioned policies. We distinguish two types of this flaw, according to which side of the security policy is not being enforced. However, as SSRs are used to bypass the SOP for CORs, SSR services suffer by design from the server-side variant of this flaw. For this reason, we focus on client-side security policies. While this flaw is not a vulnerability *per se*, as we will see in Section 4.1, insufficient client-side security policy enforcement is the root cause of a class of attacks targeting C that we call *Web Origin Laundering* attacks.

Forgery—SSR forgeries occur when S does not properly validate the user input that is used to generate the SSR, e.g., XML documents, PDF files, and URLs. SSR forgeries encompass all the currently known SSRF vulnerabilities. More specifically, this regroups and reorganizes flaws from Common Weaknesses Enumeration (CWE-113, CWE-661, CWE-829) [25], OnSec [16], and Polyakov et al. [22] which were exploited in documented attacks, i.e., against SAP NetWeaver [22], Google+ [27, 1], and Facebook [27]. Our classification also includes the TCPDF bug[3]. Besides these vulnerabilities, our classification introduces the class of *improper URL validation vulnerabilities*, which supersedes the improperly-called class of SSRF flaws (CWE-918). This group of flaws occurs when S does not validate user-provided URLs, e.g., rejecting URLs with unexpected URL schemes (e.g., `file://`), blacklisted domains, or invalid characters. Then, our classification considers two special cases of improper URL validation, i.e., improper enforcement of expected destination and improper neutralization of CRLF in HTTP headers (CWE-113). Improper enforcement of expected destination occurs when S does not sufficiently validate that the URL refers to an expected destination [25]. Improper neutralization of CRLF in HTTP headers occurs when software fails to remove the CR and LF characters from input data, such that an attacker can inject HTTP headers or *smuggle* HTTP requests.

Information Gathering—A service S can unintentionally disclose sensitive information of ES to an attacker. This class of vulnerabilities includes SSR vulnerabilities of the 2xx group of the CWE catalog, i.e., (i) improper neutralization of error messages and (ii) side channels. The former type occurs when S reveals

[3] See bug #1005, `http://sourceforge.net/projects/tcpdf/files/CHANGELOG.TXT`

information about exceptional behavior of ES in res_S. For example, S may return an error message to C detailing the reasons why ES is not reachable or the target resource is not available. Side channels occur when S unwillingly leaks information about ES. A typical side channel can be caused by a noticeable difference in the response time between $req(url_{ES})$ and res_S or by the variation in the type and size of the responses.

3.2 Behavior-Based Classification

SSRs can also be classified according to the behavior of S. We observed seven distinct behaviors that capture the way a service can be abused. While some of these may seem legitimate in isolation, we will show that their combination can lead to sophisticated attacks.

Proxy—S acts as a proxy when it returns res_{ES} to C. We distinguish proxy services as transparent (when res_{ES} is forwarded to C without any modification) or non-transparent (when, for example, res_{ES} is embedded inside res_S).

Open Origin Policy—An open origin policy service (OOP) always returns the least restrictive `ACAO:*` header, ignoring the actual value (if any) that is set by ES. OOP services allow bypassing SOP for CORs (if ES did not include the `ACAO` header) and any cross-origin resource sharing policy.

Storage—A storage service can be used to store and retrieve resources. That is, S fetches res_{ES} from ES and stores the resource locally. Then, S returns an ID to C for the resource that can be later used to retrieve res_{ES}.

Amplifier—An amplifier service can increase the number of SSRs and/or the amount of data sent in SSRs as compared to CSRs. We designate amplifiers as request amplifiers (when they increase the number of requests) or data amplifiers (when they increase the size of each request or response).

Bridge—A bridge service connects different layers of a protocol stack and allows S to send packets to non-HTTP services. With reference to Figure 1a, when S processes a crafted URL, instead of generating an HTTP request, it opens a TCP connection and sends raw data to ES. This behavior is often the result of forgery vulnerabilities, e.g., improper URL validation.

Interpreter—An interpreter service uses HTTP clients capable of interpreting JavaScript code. For instance, S can be used to control the different parts of a more complex attack, or to perform any computations on the attacker's behalf.

Probe—A probe service can be used to collect information about an external service ES. Information can be leaked to C over side channels. Depending on the information leaked, probe services can be used to perform port scanning, host discovery, or application fingerprinting. This type of service is the result of two flaws: forgery, i.e., accepting custom TCP ports, and information gathering.

Attacks		Flaw			Behaviors						Controls			Target
		No Enf.	*Forgery*	*Inf.Gath.*	*Proxy*	*OOP*	*Storage*	*Amplifier*	*Bridge*	*Interpr.*	*Probe*	*Req.Dest.*	*Req.Cont.*	*Resp.Cont.*	
Origin Laundering	Attack 1.1	✗			✗							✗		✗	C
	Attack 1.2	✗			✗							✗		✗	C
Denial of Service	Attack 2.1 (Dom. Blacklist.)	✗			✗							✗		✗	S
	Attack 2.2 (with Data Amplifier)	✗	✗				✗	✗		✗		✗	✗	✗	S, ES
	Attack 2.3 (against Data Amplifier)	✗	✗				✗	✗		✗		✗	✗	✗	S, ES
Reconnaissance Attacks			✗	✗							✗	✗			S, ES
Bridging Attacks			✗						✗			✗	✗		S, ES

Table 1: Mapping between attacks and the four angles of our classification: flaw, behavior, control, and target.

3.3 Control-Based Classification

The third SSRs classification is according to the control an attacker has on the content of SSRs and responses generated by S. In particular, we distinguish the control over the *destination* and the *content* of SSRs. The destination consists of the domain or IP address of the server, the HTTP `Host` header, and the path of the HTTP request, whereas the content of a request covers the request parameters and the body. This classification supersedes Polyakov's classification [22] as we add control over the response. For the response, we consider only the content, i.e., the body of the HTTP response res_S.

3.4 Target-Based Classification

Finally, we examine who can be the target of an SSR-based attack. We distinguish between attacks against the client C, the SSR service S, and the remote service ES. Most of the vulnerabilities discovered by prior work target ES, such as the vulnerabilities on Facebook and Google services [27], the XXE on SAP NetWeaver [22], and the vulnerability of DB4Web (CVE-2002-1484) which all allowed attacks against third-party services. We extend this threat model with attacks against the client, such as the Web Origin Laundering attack. In addition, we define S as a potential target, e.g., of resource exhaustion attacks.

4 Attacks

We now instantiate our classification and present seven attacks. Attacks are divided into four categories: browser countermeasure evasions (Section 4.1), DoS attacks (Section 4.2), reconnaissance (Section 4.3), and bridging attacks (Section 4.4). Only the last two were previously known. The mapping between attacks and our classification (including the root cause flaw) is shown in Table 1. As opposed to the known exploitations of SSRF [12, 27, 22, 15, 16], two out of seven attacks actually target C—an insight that should bring additional attention to SSR abuse.

4.1 Web Origin Laundering

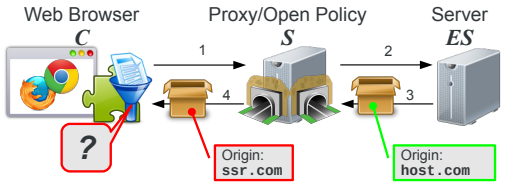


Fig. 3: The *Web Origin Laundering* attack.

Web browsers implement various URL-based defenses to protect users and data from attacks, such as Google Safe Browsing [9], NoScript [13], or AdBlock [6]. These mechanisms make security decisions based on requested URLs, e.g., limit the scope of JavaScript programs or even deny the JavaScript execution. *Web origin laundering* is an attack which hides resource origins, thus bypassing URL-based defenses, leaving users exposed. With reference to our threat model, this is an attack against C, i.e., risk **R3**. First, C requests a resource of ES via S. Note that the victim's browser is not aware of the fact that the request of step 1 contains the URL of a resource of ES. Then, S fetches the resource from ES and returns it to C (steps 2-4). Finally, the web browser verifies the origin of the resource to enforce security mechanisms. Unfortunately, the browser will falsely assume that S is the origin, possibly leading to a wrong decision in the security checks. We now preset two instances of this attack to bypass browser countermeasures.

Attack 1.1—With reference to Figure 3, the attacker prepares a URL that is distributed to C. For example, the URL refers to a proxy service to fetch malicious content hosted by ES, e.g., `http://ssr.com/?url=host.com/mal.html`. The attacker sends this URL to C, e.g., via phishing email, or linking it in forums and social networks. The victim clicks on the URL and, as a result, her browser verifies whether the URL is blacklisted. As `ssr.com` is not blacklisted, C sends message 1 to S. S extracts the URL from the parameter `url`, and fetches the malicious content at `host.com/mal.html`. Finally, it returns the malicious content to C. We have successfully performed this attack, bypassing the Google Safe Browsing mechanism as implemented by Google Chrome 43.0.2357.130 and Mozilla Firefox 39.0. In these attacks, we have used two proxy services to relay known phishing pages, drive-by download pages, and other malicious content including malware binaries (i.e., EICAR Standard Anti-Virus Test File and Virus.Win32.Virut).

Researchers have recently found criminals using a similar technique to distribute links to phishing pages. The attacker distributes a Google URL that redirected to the malicious target[4]. However, browser countermeasures can discover the attempt to redirect the user to a malicious domain and then block the attack. Furthermore, this attack is limited only to pages indexed by Google. Our attack does not rely on redirect but instead on SSRs which hide the true origin of the malicious content. Finally, an additional confirmation of the severity of

[4] See `https://isc.sans.edu/diary/How+Malware+Campaigns+Employ+Google+Redirects+and+Analytics/19843`

this threat was provided in a recent NoScript bypass attack based on a SSRF vulnerability in the content delivery network of Akamai[5].

Attack 1.2 (Escaping Content Dispositions)—Attack 1.1 can be blocked by the `Content-Disposition` response header of S. This header suggests to a browser not to display the returned resource to the user. We will discuss the use of this header in Section 6. However, even in presence of the content disposition header, it is still possible to deliver and display malicious content to the user. Consider the following JavaScript code embedded in a malicious web page hosted by a third-party service:

```
1 var malware = "http://host.com/mal.html";
2 var cor = new XMLHttpRequest();
3 cor.onreadystatechange=function() {
4  var ct = this.getResponseHeader("content-type")
5  window.location = "data:" + ct "," + encodeURIComponent(cor.responseText);
6 }
7 cor.open("GET", "http://ssr.com/?url=" + encodeURIComponent(malware), false);
8 cor.send();
```

The URL of the malicious resource, i.e., url_{ES}, is in the variable `malware`(Line 1) which is retrieved with an asynchronous request (Lines 2 and 7-8). Note that the URL used in the Ajax request is of the SSR service S (line 7). If the attacker directly used the value in `malware` (line 8), the attempt to reach a malicious server RE would be detected (e.g., by Google Safe Browsing). Then, once the malicious resource is fetched, the JavaScript program transforms it into a *data URL*. Such URL does not point to a resource, but instead contains the resource within the URL itself. Finally, the browser is directed to the data URL (line 5) and the malicious content is displayed to the user. We have developed proofs of concept of these attacks and bypassed the Google Safe Browsing mechanism of Chrome and Firefox. To this end, we used a proxy service which returned the `Content-Disposition` response header. Similarly to the previous attack, we used URLs of real phishing pages and binaries of actual malware.

4.2 Denial of Service

We now present three scenarios in which SSR is abused to perform DoS attacks against S. We group these attacks into two categories: *domain blacklisting* and *resource exhaustion*.

Attack 2.1 (DoS via Domain Blacklisting)— As discussed before, browsers prevent users from visiting websites that are known to host malicious content. An attacker may be able to poison these blacklists to block benign sites that expose a proxy behavior by using the web origin laundering technique. To this end, the attacker prepares a URL for the proxy service that contains the URL of a malicious page, and submits it to the blacklist operator (e.g., to Google in the case of Safe Browsing) to initiate a scan. Since the malicious content seems to originate from the proxy service, once the URL is detected as malicious, the proxy itself gets blacklisted. To avoid to disrupt the operations of SSR services,

[5] See https://www.blackhat.com/us-15/briefings.html

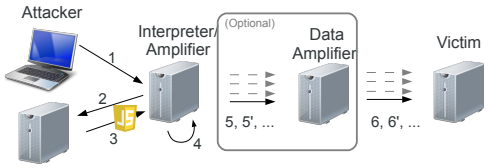


Fig. 4: DoS with Data Amplifier

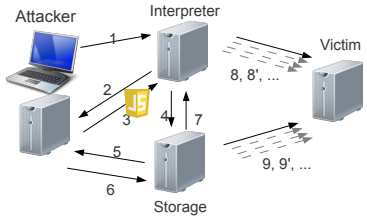


Fig. 5: DoS against Data Amplifier

we did not test this attack in practice. With reference to our threat model, this is an attack against S, i.e., risk **R4**.

Attack 2.2 (DoS with Data Amplifier): In this second scenario, an attacker can target *any* Internet-facing service and flood it with HTTP requests. The general idea is to use an interpreter service as coordinator to amplify number and size of requests by using data amplifier services via CORs. In order to bypass SOP for CORs, this attack uses the web origin laundering presented in Section 4.1 whenever the interpreter needs to send a request towards another service role. Figure 4 shows an example involving the attacker, an interpreter, and an amplifier service. The attacker (C) submits the URL of the JavaScript program to the interpreter service (S, step 1). The interpreter fetches and executes a malicious program (steps 2 and 3) that performs two operations: *enlistment* and *attack*. The enlistment consists in re-submitting the URL of the JavaScript programs to the interpreter service. This will increase the number of instances of JavaScript programs participating in the attack. In the attack phase, the JavaScript code instructs the web service to send many HTTP requests to the victim (ES). Browsers, such as used by S, can generate about 3,000 requests per second using the XMLHttpRequest API [21]. One can further increase the attack impact by using data amplifier services that receive compressed requests and submit the decompressed data to the victim (step 6). Data amplifiers allow about a 1:1000 ratio between the data sent to the amplifier and the data sent to the victim [20].

For ethical reasons we did not perform any resource exhaustion DoS attacks. Instead, we manually verified that the building blocks of this attack are offered by the services involved in the attack. More specifically, we verified that (i) interpreters offer the features needed for the attacks (e.g., XMLHttpRequest API or Image API), (ii) chains and combinations of SSR services can be created, and (iii) the composition of the services can be invoked by interpreters. With reference to our threat model, this is an attack against ES, i.e., risk **R1**.

Attack 2.3 (DoS against Data Amplifier)—A similar setup of Attack 2.2 can also be used to attack the data amplifier, by keeping it busy with decompression tasks (see Figure 5). In this case, the attack also requires a storage service to store attacker-controlled compressed data. The interpreter, again controlled by a malicious program, will request the storage service to fetch the compressed resource from the web server of the attacker (steps 4-6). Then, the storage service

returns an ID of the resource to the interpreter (step 7). Finally, the interpreter will send many compressed requests to the victim that trigger the victim to fetch resources from the storage (step 8, 8', ...). The victim is not only forced to decompress the requests, but it also has to continuously fetch compressed resources from the storage service and decompress them, easily leading to memory exhaustion. Similarly for Attack 2.2, we did not perform the attack but we verified that the building blocks of this attack are offered by the services involved in the attack. With reference to our threat model, this is an attack against S, i.e., risk **R4**.

4.3 Network Reconnaissance

Network reconnaissance is a previously-known family of attacks (i.e., risk **R1**) which entails attacks that gather information about a network, server, or service. We distinguish between *port scanning*, *host discovery*, and *application fingerprinting*. Reconnaissance is the main documented attack exploiting SSRF [15, 24]. While classical attacks require connecting directly to the victim, probe services can be used to offer anonymity and even allow access to private networks across firewall boundaries. Figure 6 shows this attack with a probe service S. The attacker prepares a request for S which contains the URL with the host or service to be scanned. For example, if the attacker would like to probe an SSH service, she can submit the URL `http://target.com:22`. As a result, S connects to the URL (an SSH server) and responds to the attacker, potentially leaking information about the status of the target service. In our example, S tries to interpret the response of the target as an HTTP response, and returns the reason for the failure (e.g., reporting that a given SSH server banner is not a valid HTTP message). If S does not leak information about the target, as we will show, an attacker can use *side channels* to determine the state of a TCP port, the availability of a resource, or the reachability of a host.

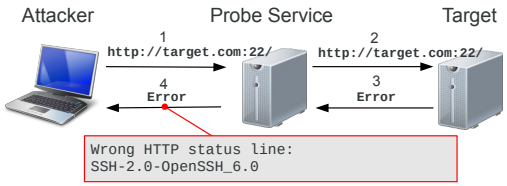


Fig. 6: Port scanning with probe services

4.4 Protocol Bridging Attacks

Protocol bridging is a previously-known family of attacks. The service S often supports different URL schemes, including `ftp`, `gopher`, and `dict`. In particular, the `gopher` scheme allows the attacker to send arbitrary data over the TCP connection, by using the following URL: `gopher://target.com:`*port*`/`*payload*. If a service does not properly validate the schemes of user-provided URLs, SSRs can be used to send arbitrary data (i.e., *payload*) over TCP connections to non-HTTP network services—effectively acting as a *bridge* between different protocols. In the past, this technique has been used to connect to remote procedure calls (RPC) services and exploit buffer overflow vulnerabilities [22], but it could

be used for many other malicious purposes, such as to send spam messages to an SMTP server. With reference to our threat model, this is an attack against ES, i.e., risk **R1**. A variation of this attack involves the `file` URL scheme to retrieve files in S's filesystem (e.g., by sending `file:///etc/passwd` to a bridge service S). In one incident, such an attack allowed access to system files (e.g., passwd) of Google servers [1]. According to our threat model, this is an attack against local resources of S, i.e., risk **R2**.

5 Case Studies and Analysis

In an attempt to investigate the prevalence of SSR attacks, we analyzed 68 services taken from seven web application classes, i.e., social networks, business web applications (e.g., spreadsheet and calendar web applications), software development tools, online image processing, OpenID service providers, RSS readers, and online web screenshot tools. For each category, we selected the most popular web applications prioritized by Google search ranks. About 60% of our case studies are among the top 50,000 web sites, including six of the top 10 web sites on Alexa.

	Flaws			Behavior					
GÜNTHER *Capabilities*	*No Enf.*	*Forgery*	*Inf. Gath.*	*Proxy*	*OOP*	*Storage*	*Amplif.*	*Bridge*	*Interpr.*	*Probe*
(T1) IP Addr. and TCP port	✗									✗
Non-HTTP schemes	✗							✗		
(T2) Transparent Proxying				✗						
Store Resource						✗				
Malicious Content	✗									
(T3) ACAO: *	✗				✗					
HTTP Req/Resp Compr.							✗			
(T4) Image/Ajax API									✗	
Web Worker							✗		✗	
(T5) Side Channel			✗							✗

Table 2: Mapping between tests and classification

The goal of our analysis is to study real SSR services and map them to our classification. To aid our analysis, we developed GÜNTHER, a novel open-source black-box testing tool[6] that reveals SSR flaws and service behaviors. GÜNTHER takes as input a description of $url(req_{ES})$, possibly enriched with session data (i.e., session cookies). Then, GÜNTHER generates a list of requests to probe the service. GÜNTHER consists of a *tester* and a *monitor* component. The tester probes S whereas the monitor dynamically spawns servers to receive SSRs originated by the service. With reference to Figure 1a, the tester and the monitor play the roles of C and S, respectively. The current version of GÜNTHER supports the tests in Table 2, i.e., **(T1)** URL validation and validation bypass via HTTP 3xx redirection, **(T2)** proxy behavior, **(T3)** response header analysis, **(T4)** HTTP client analysis, and **(T5)** side channel analysis. These tests are mapped to flaws and behaviors as shown in Table 2.

We ran GÜNTHER against the 68 services in our dataset. The experiment results are shown in Table 3. We anonymized each service in Table 3a by replacing its domain name with an identifier (column *ID*) because not all of them have

[6] The tool is freely available here: `https://github.com/tgianko/guenther`

been fixed. To improve readability, we have grouped services with the same flaws and behaviors in the same row. Our experiments revealed at least one service for each flaw and service behavior. In total, 50 out of 68 services suffer from one of the flaws in our classification. All these services are either proxy, open origin policy, probe, or bridge services. One also behaves as an amplifier and four can act as interpreters and therefore can be abused to coordinate other attacks. Then, ten services (14.7%) implement weak forms of URL validation that GÜNTHER successfully bypassed via HTTP 3xx redirections. Finally, only 14 services (20.6%) in our experiments are *not* affected by SSR-based vulnerabilities.

6 Mitigations

After discussing the vast potential and impact of SSRs, we will now discuss eight mitigations and pitfalls. From our experiments on the case studies, and reviewing the state of the art on the mitigation side, we extracted a list of seven mitigations. Finally, as none of the observed ones are sufficient to block Attack 1.2, we propose an additional mitigation to enforce URL-based browser countermeasures.

(M1) Monitoring—Monitoring is a mitigation technique which aims at detecting suspicious activity at service runtime. The owner of S5 reported to us that they rely on a sophisticated monitoring technique to detect the SSR abuse targeting C (**R3** in Figure 1c). Unfortunately, the use of monitoring to detect this type of abuse has two shortcomings which make it insufficient as a general solution. First, a complex infrastructure and a considerable amount of resources are required to support monitoring, especially for popular services that serve a large number of users. Second, while monitoring SSRs may successfully mitigate large-scale abuses, it is often ineffective for detecting low-volume attacks. For example, the advent of APT-based attacks has changed the distribution from large-scale to a targeted distribution in which only a single user or organization is attacked. For these reasons, we believe that monitoring should be complemented with further preventive guidelines.

(M2) Avoid Acting as a Proxy or Wrap Response—Among our case studies, three services can be abused as transparent proxy to serve malicious content to a client. However, we are not aware of intended use cases for transparent proxies, and thus services should be explicitly designed to avoid this behavior. For example, S can use a JSON envelope to wrap res_{ES}, which prevents a web browser from interpreting the resource res_{ES} and thus blocks the Web Origin Laundering Attack 1.1. Services S12, S59, and S60 use custom JSON data structures to wrap res_{ES}, i.e., they behaved as non-transparent proxies. However, this countermeasure alone is not sufficient to also block Attack 1.2. As this second attack uses malicious JavaScript to retrieve res_{ES}, the JavaScript program can unpack res_{ES} and then encode it as inline data (i.e., via the `data` URI scheme). Attack 1.2 can partially be mitigated by enforcing URL-based browser countermeasures, such as Google Safe Browsing, at S (see M8).

ID	Flaws			Behaviors						
	No Enf.	*Forgery*	*Inf. Gath.*	*Proxy*	*OOP*	*Storage*	*Amplif.*	*Bridge*	*Interpr.*	*Probe*
Business Applications										
S5	✗		✗	✗	✗		✗			✗
S2	✗			✗						
S1		✗	✗					✗		✗
S3, S7			✗							✗
S4, S6, S8										
Development Tools										
S10			✗		✗					✗
S12		✗	✗					✗		✗
S14		✗						✗		
S9, S11, S13			✗							✗
Image Editing										
S15	✗	✗	✗			✗		✗		✗
S16	✗		✗	✗						✗
S17			✗							✗
S18										
OpenID										
S29			✗		✗					✗
S35		✗	✗					✗		✗
S19-27, S31-34, S36-40			✗							✗
S28, S30										
RSS Readers										
S41, S46		✗	✗					✗		✗
S43-45, S47			✗							✗
S42										
Screenshot										
S54			✗		✗					✗
S53, S56			✗						✗	✗
S52, S55										
S48-51									✗	
Social Networks										
S64		✗						✗		
S67			✗						✗	✗
S59-60, S62, S65-66			✗							✗
S57-58, S61, S68, S63										
Total	4	8	47	3	4	1	1	8	7	47

(a) Case studies to flaws/behaviors

Tests		Accept	Reject	Bypass
(T1)	IP address	60	8	2
	Dict scheme	4	64	4
	Goph. scheme	3	65	4
	TCP port	55	13	0

(b) URL validation results

Tests		Serv.	% rel.
(T2)	Transp. Prox.	3	4.41%
	Non-transp. Prox.	6	8.82%
	Store Resource	1	1.47%
	Malicious URL	4	5.88%
(T3)	ACAO:*	4	5.88%
	Decompr. Req.	1	1.47%
	Decompr. Resp.	36	52.94%
(T4)	Image API	7	10.29%
	XMLHTTPReq. API	5	7.35%
	Web Worker	2	2.94%

(c) Proxy, header, and client test results

(T5) Tests	Serv.	% rel.
Port Scanning		
Open/closed/filtered	13	19.12%
Open (partially)	40	58.82%
No leak	15	22.06%
Fingerprinting		
Res. exists/does not exist	37	54.41%
Res. exists (partially)	3	4.41%
No leak	24	41.18%
Host Discovery		
On/offline	45	66.18%
Online (partially)	16	23.53%
No leak	7	10.29%

(d) Side channel analysis

Table 3: Results of our Experimental Analysis

(M3) Perform Proper URL Validation—S should validate url_{ES} before fetching the target resource. Table 3b shows how our case studies validate user-provided URLs. The vast majority accept URLs containing an IP address (60 services) and/or a port number (55 services). None of these behaviors can be considered a vulnerability per se. Some applications rejected URLs with IP addresses, probably as an attempt to block attackers who may try to access local machines in the company intranet. However, it is important to understand that this countermeasure is often insufficient, as attackers can still address any IP by pointing an attacker-controlled domain to a local IP address (*DNS rebinding*). Moreover, we found weak forms of URL validation that can be circumvented. URL validation of ten services can be bypassed with HTTP redirections (last

column in Table 3b). This is critical, because it shows how the service developers attempted to mitigate the problem, but were not aware of all the details of this security threat. Worse, while few of the 68 services accept URLs with the Dict (four services) or Gopher schemes (three services), redirection helps to bypass an additional four cases for each scheme. These *bridges* are a severe threat, as they give full control of a TCP socket and enable attackers to communicate with non-HTTP network services.

URL validation that protects against rebinding can be implemented in HTTP libraries. To the best of our knowledge, SafeCurl [14] is the only HTTP library that provides these countermeasures for PHP services. Developers using other programming languages or headless browsers need to implement the above mechanisms on their own.

(M4) Content Disposition—The content disposition header is used to suggest that a browser should not display a resource inline [7]. This header has been proposed in the past to fix Reflected File Download attacks [11]. An SSR service that uses this header can block the Web Origin Laundering Attack 1.1. In fact, as the resource is not shown to the user inline, phishing attacks are prevented. In our experiments, services S5 and S9 use the content disposition header. While Content-Disposition mitigates Attack 1.1, it does not protect from Attack 1.2. Content-Disposition alone does not solve the root cause of the insufficient security policy enforcement flaw, but instead raises the difficulty for an attacker to abuse SSRs. To mitigate Attack 1.2, see M8.

(M5) Limit Resource Usage—DoS attacks of Section 4.2 are the result of a combination of services: interpreters to orchestrate the attack, amplifiers to amplify the size and number of requests, and OOP services to chain SSRs services. This mitigation targets the first two services (for the OOP services, see instead **M6**). Table 3c shows that 10% of our case studies use browsers with full JavaScript support, including JavaScript APIs that can be used to orchestrate DoS attacks. In particular, seven services support the Image API, five services support the XMLHttpRequest API, and two services support the Web Worker API. These APIs can be abused to turn a seemingly innocuous web browser into a weaponized HTTP-based bot that can generate thousands of HTTP requests per second [10, 21]. To avoid this abuse, interpreter services need to limit the request rate. Another source of resource exhaustion is data compression. With reference to Table 3c, data compression is supported by most tested services, and one also supports HTTP request decompression. Decompressing HTTP requests is not a standardized behavior, but instead is a web server-specific feature [20]. We are not aware of the reasons to support this feature, and we would recommend disabling it. Unlike this particular case, HTTP response compression is standardized and a more common feature. Also in this case, we would recommend disabling data compression. If this is not possible, then developers should verify that their services limit the resources used when decompressing incoming messages (see [20] for guidelines toward a secure implementation of data decompression).

(M6) Remove Open Access Control Policies for CORs—As OOP services can be accessed via CORs from any domain, they can be used by interpreters to chain SSR services in order to mount the attack. Among our case studies, four services use the header `ACAO: *`, which is bad practice in the presence of our threat model. The other 64 services omit ACAO headers, thus effectively blocking cross-origin requests. Another effective countermeasure to block this attack is to limit the access to SSR services to CORs from trusted origins.

(M7) Limit Information Leakage—72% of the services can be used as probes to perform network reconnaissance. This makes this role the most widespread behavior among the applications we tested. All probe services of Table 3a allow, with different degrees of granularity, network reconnaissance via response time analysis and response code. Information leaks can be solved by making S's behavior independent from the success of the SSR. For example, S can enforce a constant response time (i.e., a fixed delay between C's request and the response sent to C). We observed this behavior for 15 services that do not allow distinguishing the port state, seven services that do not leak information about the host availability, and 24 services that do not disclose the availability of an HTTP resource. However, enforcing a constant time introduces undesired delays, thus making it unsustainable for scenarios in which responsiveness is important. In these cases, S may deploy weaker security measures which can limit network reconnaissance attacks. This can be achieved, for example, by accepting URLs only with selected TCP ports with mitigation M3, or by rate-limiting the requests.

(M8) Enforce URL-Based Browser Countermeasures—None of the mitigations we obseved in the wild (M1-7) can solve Web Origin Laundering Attack 1.2. The root cause of this attack is that S allows one to retrieve and serve malicious content, and hide the true origin of the malicious content with S's domain.

To block Web Origin Laundering Attack 1.2, we propose that SSR services should implement the same countermeasures deployed by browsers in order to block harmful and unwanted content, e.g., Google Safe Browsing. Once the client submits the URL to S, S validates the URL using the Google Safe Browsing protocol. If the URL is malicious, then S refuses to retrieve it. While this approach at least partially mitigates the distribution of malicious content, it does not fix the problem if web browsers implement custom security policies, e.g., NoScript or AdBlock custom domain blacklists. In conclusion, a general solution to Web Origin Laundering Attack 1.2 is still lacking.

7 Developers Feedback

We responsibly disclosed all vulnerabilities to the respective developers. In most of the cases, developers reacted to our first reports. If developers were unresponsive for over a month, we tried a second time and then alerted the US CERT. Our disclosure resulted in a variety of responses from developers, strongly related to the type of flaw of our classification.

Forgery—75% of such vulnerabilities have been fixed by now. Six vendors (i.e., S1, S14, S15, S35, S46, and S64) patched their services, while two vendors (S12 and S41) were unresponsive. The high number of fixes may be due to a partial awareness of the security risks of forgery vulnerabilities: forgery is the first documented SSR flaw, and developers deploy countermeasures against forgery, i.e., URL validation (e.g., 64 services reject URLs with non-HTTP schemes, 13 with TCP ports, and eight with IPs). However, the fact that countermeasures can be bypassed with HTTP redirections indicates that the complete exploitation space of SSR flaws is not entirely understood.

Information gathering—The disclosure of these vulnerabilities revealed a more fragmented situation. Five services patched the problem, while the vast majority ignored the issue or did not respond to our report. An interpretation of these results is in the rejected reports. In three cases, developers did not want to modify S as they are using monitoring to prevent abuses (i.e., S3, S5, and S59). The use of monitoring suggests prudence and a general attention to security-related issues. However, the choice of monitoring over a patch in S may indicate that developers rate this risk a low priority. Other developers (S7, S60, and S62) consider this flaw not to be a security risk at all.

Enforcement of security policies—Out of four affected services, S15 has been shut down and S2 has partially solved the flaw by adding the content disposition header into the response. Developers of S16 reported having fixed the flaw, but the patch did not solve the problem. Lastly, developers of S5 rejected our report because they use monitoring to prevent abuses. As discussed in Section 6, monitoring may work for large-scale abuses, but potentially still misses individual exploitations.

8 Related Work

In this section, we review SSR literature according to four thematic groups. First, we review academic literature with a focus on vulnerability analysis and detection. Then, we review known SSR-based attacks against popular web applications. Third, we present current attempts to classify and categorize existing SSR threats. Finally, we survey existing tools to detect SSR vulnerabilities.

Vulnerability Analyses and Detection—Web vulnerabilities have been extensively studied from different angles, e.g., categorization and prioritization [17, 23], impact and trends [18], detection techniques [2, 19] and effectiveness [4], and defense mechanisms [3]. While existing works focused largely on classical, yet severe, vulnerabilities, to the best of our knowledge, no scientific work has studied the SSR communication pattern.

Attacks and Classifications—The vast majority of security incidents are described in reports and whitepapers. These attacks are SSR forgery attacks and were brought to the community's attention by Polyakov et al. [22] and Walikar [27]. Polyakov et al. [22] described an XXE vulnerability in SAP NetWeaver whereas Walikar [27] described an insufficient input validation vulnerability in popular social networks. Other exploitations of SSR forgery vulnerabilities were

reported by Almroth et al. [1], in which they retrieve local resources in Google services. All these attacks are included in Polyakov's threat model. With respect to the current knowledge about SSR-based attacks, our paper presents five previously-unknown SSR-based attacks, i.e., two Web Origin Laundering attacks and three DoS attacks.

Following the initial incidents, the community started classifying and categorizing known SSR-based vulnerabilities. All efforts focused on SSR forgery (e.g., CWE [25] and OnSec [16]). However, current knowledge on SSR vulnerabilities is sparse, disjoint, and incomplete. While the CWE database includes some SSR-related vulnerabilities, they are mainly isolated entries which are not correlated to each other. As a result, developers cannot identify all possible flaws that can affect an SSR service. Furthermore, as we have shown, there are other attacks targeting C and S which do not rely on forgery but instead abuse improper enforcement of security policies.

Detection Tools—Existing detection tools target only SSRF vulnerabilities. They are available in the form of proof-of-concept scripts (e.g., the SSRF bible [16]) or as testing tools. A proprietary tool that can find SSRF vulnerabilities is Acunetix WVS version 9 with AcuMonitor[7]. However, this tool is not freely available and we were not able to inspect it. Existing public tools offer limited detection power (only SSRF) which make them inapplicable to the purpose of this paper. Ussrfuzzer [28] fuzzes HTTP requests with URLs to detect SSRs, however, it does not perform any security test. In contrast, the OWASP Skanda [5] tool can detect information disclosure flaws, in particular leaks of TCP port status. However, it cannot be used to detect other types of leakage, e.g., web application fingerprint, nor other vulnerabilities or security related features. For all these reasons, we developed `GÜNTHER`, a first comprehensive SSR testing tool, that we plan to release to the public.

9 Conclusion

To the best of our knowledge, this is the first comprehensive study of the security of SSRs. We presented a classification of SSRs based on the type of flaw, the level of control of the messages, the behavior of the vulnerable services, and the potential attack targets. Furthermore, we unveiled previously-unknown exploitations techniques in which a combination of seemingly innocuous services can be used to mount sophisticated attacks targeting both users and servers on the Internet. We also presented experiments on 68 popular web applications. Our experiments showed that the majority of the web applications can be abused to perform malicious activities, ranging from server-side code execution to DoS attacks. We also presented eight mitigations to help developers to implement SSRs in a more secure way.

[7] See `http://www.acunetix.com/vulnerability-scanner/`

Acknowledgments

This work was supported by the German Federal Ministry of Education and Research (BMBF) through funding for the Center for IT-Security, Privacy and Accountability (CISPA) and for the BMBF project 13N13250.

References

1. F. Almroth and M. Karlsson, "How we got read access on Googles production servers." [Online]. Available: http://blog.detectify.com/post/82370846588/how-we-got-read-access-on-googles-production
2. D. Balzarotti, M. Cova, V. V. Felmetsger, and G. Vigna, "Multi-module vulnerability analysis of web-based applications," in *ACM CCS '07*.
3. A. Barth, C. Jackson, and J. C. Mitchell, "Robust defenses for cross-site request forgery," in *ACM CCS '08*.
4. J. Bau, E. Bursztein, D. Gupta, and J. Mitchell, "State of the art: Automated black-box web application vulnerability testing," in *IEEE S&P '10*.
5. J. Chauhan, "OWASP SKANDA - SSRF Exploitation Framework." [Online]. Available: http://www.chmag.in/article/may2013/owasp-skanda-%E2%80%93-ssrf-exploitation-framework
6. Eyeo GmbH, "Adblock Plus." [Online]. Available: https://adblockplus.org/
7. R. Fielding, J. Gettys, J. Mogul, H. Frystyk, L. Masinter, P. Leach, and T. Berners-Lee, "Hypertext Transfer Protocol – HTTP/1.1," RFC 2616.
8. B. Fitzpatrick, D. Recordon, D. Hardt, and J. Hoyt, "OpenID Authentication 2.0 - Final." [Online]. Available: http://openid.net/specs/openid-authentication-2_0.html
9. Google Inc., "Safe Browsing API." [Online]. Available: https://developers.google.com/safe%2Dbrowsing/
10. J. Grossman and M. Johansen, "Million Browser Botnet." [Online]. Available: https://media.blackhat.com/us%2D13/us%2D13%2DGrossman%2DMillion%2DBrowser%2DBotnet.pdf
11. O. Hafif, "Reflected File Download a New Web Attack Vector." [Online]. Available: https://drive.google.com/file/d/0B0KLoHg_gR_XQnV4RVhlNl96MHM/view
12. D. Heiland, "Web Portals Gateway to Information or a Hole in our Perimeter Defenses." [Online]. Available: http://www.shmoocon.org/2008/presentations/Web+portals,+gateway+to+information.ppt
13. InformAction, "NoScript." [Online]. Available: https://noscript.net/
14. Jack Whitton, "SafeCurl." [Online]. Available: https://github.com/fin1te/safecurl
15. P. Kulkarni, "SSRF/XSPA Bug in https://www.coinbase.com)," 06. [Online]. Available: http://www.prajalkulkarni.com/2013/06/ssrfxspa%2Dbug%2Din%2Dhttpswwwcoinbasecom.html
16. ONsec Lab, "SSRF Bible. Cheatsheet." [Online]. Available: https://docs.google.com/document/d/1v1TkWZtrhzRLy0bYXBcdLUedXGb9njTNIJXa3u9akHM
17. OWASP, "The OWASP Top 10 Project." [Online]. Available: https://www.owasp.org/index.php/Category:OWASP_Top_Ten_Project
18. P. Payet, A. Doupé, C. Kruegel, and G. Vigna, "Ears in the wild: large-scale analysis of execution after redirect vulnerabilities," in *ACM SAC '13*.
19. G. Pellegrino and D. Balzarotti, "Toward black-box detection of logic flaws in web applications," in *NDSS '14*.

20. G. Pellegrino, D. Balzarotti, S. Winter, and N. Suri, "In the Compression Hornet's Nest: A Security Study of Data Compression in Network Services," in *USENIX Security '15*.
21. G. Pellegrino, C. Rossow, F. J. Ryba, T. C. Schmidt, and M. Wählisch, "Cashing Out the Great Cannon? On Browser-Based DDoS Attacks and Economics," in *USENIX WOOT '15*.
22. A. Polyakov, D. Chastukjin, and A. Tyurin, "SSRF vs. Business-Critical Applications. Part 1: XXE Tunnelling in SAP NetWeaver." [Online]. Available: http://erpscan.com/wp%2Dcontent/uploads/2012/08/SSRF%2Dvs%2DBusinness%2Dcritical%2Dapplications%2Dwhitepaper.pdf
23. SANS Institute, "Critical Security Controls for Effective Cyber Defense." [Online]. Available: https://www.sans.org/media/critical-security-controls/CSC-5.pdf
24. A. Santese, "Yahoo! SSRF/XSPA Vulnerability)," 06. [Online]. Available: http://hacksecproject.com/yahoo%2Dssrfxspa%2Dvulnerability/
25. The MITRE Corporation, "Common Weakness Enumeration." [Online]. Available: http://http://cwe.mitre.org/
26. A. van Kesteren, "Cross-Origin Resource Sharing - W3C Recommendation 16 January 2014." [Online]. Available: http://www.w3.org/TR/cors/
27. R. A. Walikar, "Cross Site Port Attacks - XSPA." [Online]. Available: http://www.riyazwalikar.com/2012/11/cross%2Dsite%2Dport%2Dattacks%2Dxspa%2Dpart%2D1.html
28. E. Zaitov, "Universal SSRF Fuzzer." [Online]. Available: https://github.com/kyprizel/ussrfuzzer