21

INFORMATION DISCLOSURE

The IDOR vulnerabilities covered in Chapter 10 are a common way for applications to leak private information about users. But an attacker can uncover sensitive information from a target application in other ways too. I call these bugs *information disclosure* bugs. These bugs are common; in fact, they're the type of bug I find most often while bug bounty hunting, even when I'm searching for other bug types.

These bugs can happen in many ways, depending on the application. In this chapter, we'll talk about a few ways you might manage to leak data from an application, and how you can maximize the chances of finding an information disclosure yourself. This chapter delves into some of the techniques mentioned in Chapter 5, but with a focus on extracting sensitive and private information by using these techniques.

Mechanisms

Information disclosure occurs when an application fails to properly protect sensitive information, giving users access to information they shouldn't have available to them. This sensitive information can include technical details that aid an attack, like software version numbers, internal IP addresses, sensitive filenames, and filepaths. It could also include source code that allows attackers to conduct a source code review on the application. Still other times, the application leaks private information of users, like a user's age, bank account numbers, email addresses, and mailing addresses, to unauthorized third parties.

Most systems aim to hide development information, including software version numbers and configuration files, from the outside world, because it allows attackers to gather information about an application and strategize about how to most effectively attack it. For example, learning the exact software versions an application uses will allow attackers to look for publicly disclosed vulnerabilities that affect the application. Configuration files often contain information such as access tokens and internal IP addresses that attackers can use to further compromise the organization.

Typically, applications leak version numbers in HTTP response headers, HTTP response bodies, or other server responses. For example, the `X-Powered-By` header, which is used by many applications, shows you which framework the application runs:

```
X-Powered-By: PHP/5.2.17
```

On the other hand, applications leak sensitive configuration files by not applying proper access control to the files, or by accidentally uploading a sensitive file onto a public repository that outside users can access.

Another piece of information that applications should protect is their source code. When the backend code of an application is leaked to the public, the leaked code can help attackers understand the application's logic, as well as search for logic flaw vulnerabilities, hardcoded credentials, or information about the company's infrastructure, such as internal IPs. Applications can leak source code by accidentally publishing a private code repository, by sharing code snippets on public GitHub or GitLab repositories, or by uploading it to third-party sites like Pastebin.

Finally, applications often leak sensitive information by including it in their public code. Developers might accidentally place information such as credentials, internal IP addresses, informative code comments, and users' private information in public source code such as the HTML and JavaScript files that get served to users.

Prevention

It's difficult to completely prevent sensitive information leaks. But you can reliably lower the possibilities of information disclosure by safeguarding your data during the development process.

The most important measure you should take is to avoid hardcoding credentials and other sensitive information into executable code. Instead, you can place sensitive information in separate configuration files or a secret storage system like Vault (*https://github.com/hashicorp/vault/*). Also, audit your public code repositories periodically to make sure sensitive files haven't been uploaded by accident. Tools can help you monitor code for secrets, such as secret-bridge (*https://github.com/duo-labs/secret-bridge/*). And if you have to upload sensitive files to the production server, apply granular access control to restricts users' access to the files.

Next, remove data from services and server responses that reveals technical details about the backend server setup and software versions. Handle all exceptions by returning a generic error page to the user, instead of a technical page that reveals details about the error.

Hunting for Information Disclosure

You can use several strategies to find information disclosure vulnerabilities, depending on the application you're targeting and what you're looking for. A good starting point is to look for software version numbers and configuration information by using the recon techniques introduced in Chapter 5. Then you can start to look for exposed configuration files, database files, and other sensitive files uploaded to the production server that aren't protected. The following steps discuss some techniques you can attempt.

Step 1: Attempt a Path Traversal Attack

Start by trying a path traversal attack to read the server's sensitive files. *Path traversal attacks* are used to access files outside the web application's root folder. This process involves manipulating filepath variables the application uses to reference files by adding the `../` characters to them. This sequence refers to the parent directory of the current directory in Unix systems, so by adding it to a filepath, you can often reach files outside the web root.

For example, let's say a website allows you to load an image in the application's image folder by using a relative URL. An *absolute* URL contains an entire address, from the URL protocol to the domain name and pathnames of the resource. *Relative* URLs, on the other hand, contain only a part of the full URL. Most contain only the path or filename of the resource. Relative URLs are used to link to another location on the same domain.

This URL, for example, will redirect users to *https://example.com/images/1.png*:

```
https://example.com/image?url=/images/1.png
```

In this case, the `url` parameter contains a relative URL (*/images/1.png*) that references files within the web application root. You can insert the `../` sequence to try to navigate out of the images folder and out of the web root.

For instance, the following URL refers to the *index.html* file at the web application's root folder (and out of the *images* folder):

```
https://example.com/image?url=/images/../index.html
```

Similarly, this one will access the */etc/shadow* file at the server's root directory, which is a file that stores a list of the system's user accounts and their encrypted passwords:

```
https://example.com/image?url=/images/../../../../../../../etc/shadow
```

It might take some trial and error to determine how many `../` sequences you need to reach the system's root directory. Also, if the application implements some sort of input validation and doesn't allow `../` in the filepath, you can use encoded variations of `../`, such as `%2e%2e%2f` (URL encoding), `%252e%252e%255f` (double URL encoding), and `..%2f` (partial URL encoding).

Step 2: Search the Wayback Machine

Another way to find exposed files is by using the Wayback Machine. Introduced in Chapter 5, the Wayback Machine is an online archive of what websites looked like at various points in time. You can use it to find hidden and deprecated endpoints, as well as large numbers of current endpoints without actively crawling the site, making it a good first look into what the application might be exposing.

On the Wayback Machine's site, simply search for a domain to see its past versions. To search for a domain's files, visit *https://web.archive.org/web/*/DOMAIN*.

Add a /* to this URL to get the archived URLs related to the domain as a list. For example, *https://web.archive.org/web/*/example.com/** will return a list of URLs related to *example.com*. You should see the URLs displayed on the Wayback Machine web page (Figure 21-1).

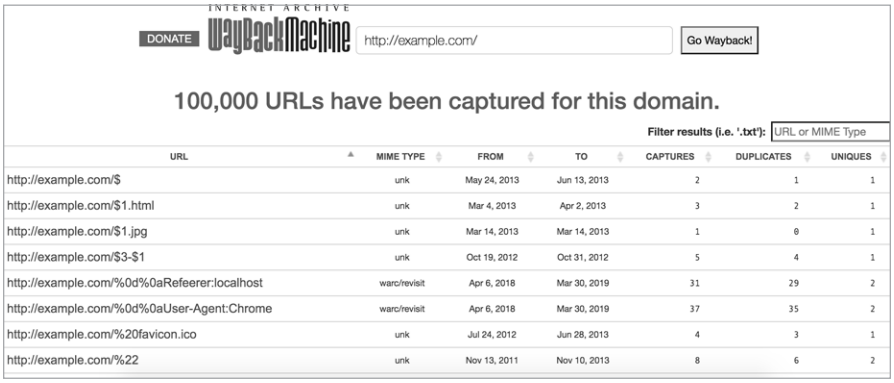

Figure 21-1: You can list the archived URLs of a domain on the Wayback Machine.

You can then use the search function to see whether any sensitive pages have been archived. For example, to look for admin pages, search for the term */admin* in the found URLs (Figure 21-2).

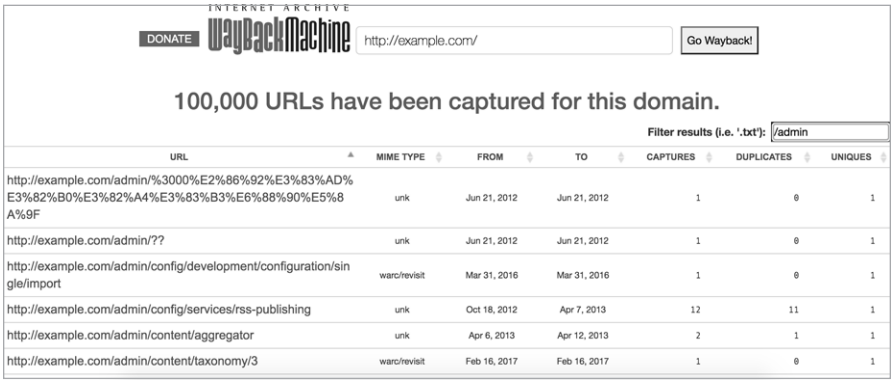

Figure 21-2: Search for keywords in the URLs to find potentially sensitive pages.

You can also search for backup files and configuration files by using common file extensions like *.conf* (Figure 21-3) and *.env*, or look for source code, like JavaScript or PHP files, by using the file extensions *.js* and *.php*.

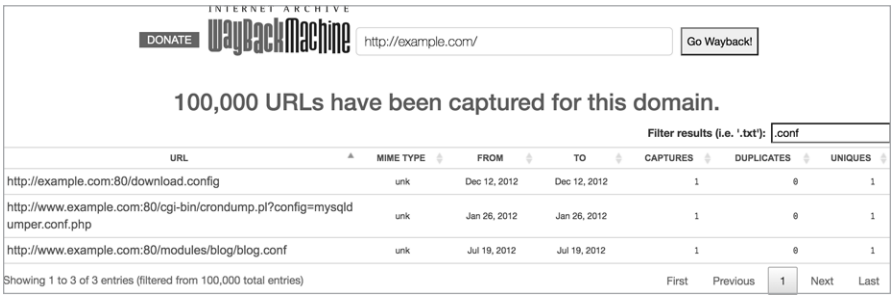

Figure 21-3: Filter the URLs by file extension to find files of a certain type.

Download interesting archived pages and look for any sensitive info. For example, are there any hardcoded credentials that are still in use, or does the page leak any hidden endpoints that normal users shouldn't know about?

Step 3: Search Paste Dump Sites

Next, look into paste dump sites like Pastebin and GitHub gists. These let users share text documents via a direct link rather than via email or services like Google Docs, so developers often use them to send source code, configuration files, and log files to their coworkers. But on a site like Pastebin, for example, shared text files are public by default. If developers upload a sensitive file, everyone will be able to read it. For this reason, these code-sharing sites are pretty infamous for leaking credentials like API keys and passwords.

Pastebin has an API that allows users to search for public paste files by using a keyword, email, or domain name. You can use this API to find sensitive files that belong to a certain organization. Tools like PasteHunter or pastebin-scraper can also automate the process. Pastebin-scraper (*https://github.com/streaak/pastebin-scraper/*) uses the Pastebin API to help you search for paste files. This tool is a shell script, so download it to a local directory and run the following command to search for public paste files associated with a particular keyword. The `-g` option indicates a general keyword search:

```
./scrape.sh -g KEYWORD
```

This command will return a list of Pastebin file IDs associated with the specified `KEYWORD`. You can access the returned paste files by going to *pastebin.com/ID*.

Step 4: Reconstruct Source Code from an Exposed .git Directory

Another way of finding sensitive files is to reconstruct source code from an exposed *.git* directory. When attacking an application, obtaining its source code can be extremely helpful for constructing an exploit. This is because some bugs, like SQL injections, are way easier to find through static code analysis than black-box testing. Chapter 22 covers how to review code for vulnerabilities.

When a developer uses Git to version-control a project's source code, Git will store all of the project's version-control information, including the commit history of project files, in a Git directory. Normally, this *.git* folder shouldn't be accessible to the public, but sometimes it's accidentally made available. This is when information leaks happen. When a *.git* directory is exposed, attackers can obtain an application's source code and therefore gain access to developer comments, hardcoded API keys, and other sensitive data via secret scanning tools like truffleHog (*https://github.com/dxa4481/truffleHog/*) or Gitleaks (*https://github.com/zricethezav/gitleaks/*).

Checking Whether a .git Folder Is Public

To check whether an application's *.git* folder is public, simply go to the application's root directory (for example, *example.com*) and add */.git* to the URL:

```
https://example.com/.git
```

Three things could happen when you browse to the */.git* directory. If you get a 404 error, this means the application's *.git* directory isn't made available to the public, and you won't be able to leak information this way. If you get a 403 error, the *.git* directory is available on the server, but you won't be able to directly access the folder's root, and therefore won't be able to list all the files contained in the directory. If you don't get an error and the server responds with the directory listing of the *.git* directory, you can directly browse the folder's contents and retrieve any information contained in it.

Downloading Files

If directory listing is enabled, you can browse through the files and retrieve the leaked information. The `wget` command retrieves content from web servers. You can use `wget` in recursive mode (`-r`) to mass-download all files stored within the specified directory and its subdirectories:

```
$ wget -r example.com/.git
```

But if directory listing isn't enabled and the directory's files are not shown, you can still reconstruct the entire *.git* directory. First, you'll need to confirm that the folder's contents are indeed available to the public. You can do this by trying to access the directory's *config* file:

```
$ curl https://example.com/.git/config
```

If this file is accessible, you might be able to download the Git directory's entire contents so long as you understand the general structure of *.git* directories. A *.git* directory is laid out in a specific way. When you execute the following command in a Git repository, you should see contents resembling the following:

```
$ ls .git
COMMIT_EDITMSG HEAD branches config description hooks index info logs objects refs
```

The output shown here lists a few standard files and folders that are important for reconstructing the project's source. In particular, the */objects* directory is used to store Git objects. This directory contains additional folders; each has two character names corresponding to the first two characters of the SHA1 hash of the Git objects stored in it. Within these subdirectories, you'll find files named after the rest of the SHA1 hash of the Git object stored in it. In other words, the Git object with a hash of `0a082f2656a655c8b0a87956c7bcdc93dfda23f8` will be stored with the filename of *082f2656a655c8b0a87956c7bcdc93dfda23f8* in the directory *.git/objects/0a*. For example, the following command will return a list of folders:

```
$ ls .git/objects
00 0a 14 5a 64 6e 82 8c 96 a0 aa b4 be c8 d2 dc e6 f0 fa info pack
```

And this command will reveal the Git objects stored in a particular folder:

```
$ ls .git/objects/0a
082f2656a655c8b0a87956c7bcdc93dfda23f8 4a1ee2f3a3d406411a72e1bea63507560092bd 66452433322af3d3
19a377415a890c70bbd263 8c20ea4482c6d2b0c9cdaf73d4b05c2c8c44e9 ee44c60c73c5a622bb1733338d3fa964
b333f0
0ec99d617a7b78c5466daa1e6317cbd8ee07cc 52113e4f248648117bc4511da04dd4634e6753
72e6850ef963c6aeee4121d38cf9de773865d8
```

Git stores different types of objects in *.git/objects*: commits, trees, blobs, and annotated tags. You can determine an object's type by using this command:

```
$ git cat-file -t OBJECT-HASH
```

Commit objects store information such as the commit's tree object hash, parent commit, author, committer, date, and message of a commit. *Tree* objects contain the directory listings for commits. *Blob* objects contain copies of files that were committed (read: actual source code!). Finally, *tag* objects contain information about tagged objects and their associated tag names. You can display the file associated with a Git object by using the following command:

```
$ git cat-file -p OBJECT-HASH
```

The */config* file is the Git configuration file for the project, and the */HEAD* file contains a reference to the current branch:

```
$ cat .git/HEAD
ref: refs/heads/master
```

If you can't access the */.git* folder's directory listing, you have to download each file you want instead of recursively downloading from the directory root. But how do you find out which files on the server are available when object files have complex paths, such as *.git/objects/0a/72e6850ef963c6aeee4121d38cf9de773865d8*?

You start with filepaths that you already know exist, like *.git/HEAD*! Reading this file will give you a reference to the current branch (for example, *.git/refs/heads/master*) that you can use to find more files on the system:

```
$ cat .git/HEAD
ref: refs/heads/master
$ cat .git/refs/heads/master
0a66452433322af3d319a377415a890c70bbd263
$ git cat-file -t 0a66452433322af3d319a377415a890c70bbd263
commit
$ git cat-file -p 0a66452433322af3d319a377415a890c70bbd263
tree 0a72e6850ef963c6aeee4121d38cf9de773865d8
```

The *.git/refs/heads/master* file will point you to the particular object hash that stores the directory tree of the commit. From there, you can see that the object is a commit and is associated with a tree object, `0a72e6850ef963c6aeee4121d38cf9de773865d8`. Now examine that tree object:

```
$ git cat-file -p 0a72e6850ef963c6aeee4121d38cf9de773865d8
100644 blob 6ad5fb6b9a351a77c396b5f1163cc3b0abcde895 .gitignore
040000 blob 4b66088945aab8b967da07ddd8d3cf8c47a3f53c source.py
040000 blob 9a3227dca45b3977423bb1296bbc312316c2aa0d README
040000 tree 3b1127d12ee43977423bb1296b8900a316c2ee32 resources
```

Bingo! You discover some source code files and additional object trees to explore.

On a remote server, your requests to discover the different files would look a little different. For instance, you can use this URL to determine the HEAD:

```
https://example.com/.git/HEAD
```

Use this URL to find the object stored in that HEAD:

```
https://example.com/.git/refs/heads/master
```

Use this URL to access the tree associated with the commit:

```
https://example.com/.git/objects/0a/72e6850ef963c6aeee4121d38cf9de773865d8
```

Finally, use this URL to download the source code stored in the *source.py* file:

```
https://example.com/.git/objects/4b/66088945aab8b967da07ddd8d3cf8c47a3f53c
```

If you are downloading files from a remote server, you'll also need to decompress the downloaded object file before you read it. This can be done using some code. You can decompress the object file by using Ruby, Python, or your preferred language's *zlib* library:

```
ruby -rzlib -e 'print Zlib::Inflate.new.inflate(STDIN.read)' < OBJECT_FILE

python -c 'import zlib, sys;
  print repr(zlib.decompress(sys.stdin.read()))' < OBJECT_FILE
```

After recovering the project's source code, you can `grep` for sensitive data such as hardcoded credentials, encryption keys, and developer comments. If you have time, you can browse through the entire recovered codebase to conduct a source code review and find potential vulnerabilities.

Step 5: Find Information in Public Files

You could also try to find information leaks in the application's public files, such as their HTML and JavaScript source code. In my experience, JavaScript files are a rich source of information leaks!

Browse the web application that you're targeting as a regular user and take note of where the application displays or uses your personal information. Then right-click those pages and click **View page source**. You should see the HTML source code of the current page. Follow the links on this page to find other HTML files and JavaScript files the application is using. Then, on the HTML file and the JavaScript files found, `grep` every page for hardcoded credentials, API keys, and personal information with keywords like `password` and `api_key`.

You can also locate JavaScript files on a site by using tools like LinkFinder (*https://github.com/GerbenJavado/LinkFinder/*).

Escalating the Attack

After you've found a sensitive file or a piece of sensitive data, you'll have to determine its impact before reporting it. For example, if you have found credentials such as a password or an API key, you need to validate that they're currently in use by accessing the target's system with them. I often find outdated credentials that cannot be used to access anything. In that case, the information leak isn't a vulnerability.

If the sensitive files or credentials you've found are valid and current, consider how you can compromise the application's security with them. For example, if you found a GitHub access token, you can potentially mess with the organization's projects and access their private repositories. If you find the password to their admin portals, you might be able to leak their customers' private information. And if you can access the */etc/shadow* file on a target server, you might be able to crack the system user's passwords and take over the system! Reporting an information leak is often about communicating the impact of that leak to companies by highlighting the criticality of the leaked information.

If the impact of the information you found isn't particularly critical, you can explore ways to escalate the vulnerability by chaining it with other security issues. For example, if you can leak internal IP addresses within the target's network, you can use them to pivot into the network during an SSRF exploit. Alternatively, if you can pinpoint the exact software version numbers the application is running, see if any CVEs are related to the software version that can help you achieve RCE.

Finding Your First Information Disclosure!

Now that you understand the common types of information leaks and how to find them, follow the steps discussed in this chapter to find your first information disclosure:

1. Look for software version numbers and configuration information by using the recon techniques presented in Chapter 5.
2. Start searching for exposed configuration files, database files, and other sensitive files uploaded to the production server that aren't protected properly. Techniques you can use include path traversal, scraping the Wayback Machine or paste dump sites, and looking for files in exposed *.git* directories.
3. Find information in the application's public files, such as its HTML and JavaScript source code, by `grep`ping the file with keywords.
4. Consider the impact of the information you find before reporting it, and explore ways to escalate its impact.
5. Draft your first information disclosure report and send it over to the bug bounty program!