**Information disclosure**

* **Introduction**
  + **What is Sensitive Data Exposure?**
  + **information disclosure Impact**
  + **How to test for information disclosure vulnerabilities**
  + **Data Leaks What Information Matters?**
  + **Data Leaks – Low Data Values**
* **COMMON Sources of Source code disclosure**
  + **Information disclosure through Developers comments**
  + **Files for web crawlers**
  + **File Name & File Path Disclosure**
  + **Information Disclosure Through Directory Listing**
  + **Brute forcing web application directories/files**
  + **Information Disclosure Through ERROR MESSAGES**
  + **Source code disclosure via backup files (.old/.bak/.php~)**
  + **Checking JavaScript files for useful URLs / endpoints**
  + **Information disclosure through git directories / GitHub repository**
    - **Git directories**
    - **Git repository**
  + **Information disclosure due to insecure configuration**
* **Preventing Data Exposure**
* **Cross Site Script Inclusion (XSSI)**

**----------------------------------------------------------------------------------------------------**

* **Introduction**
  + **What is Sensitive Data Exposure?**
    - information disclosure, also known as information leakage, is when a website unintentionally reveals sensitive information to its users. Depending on the context, websites may leak all kinds of information to a potential attacker, including:
      * Data about other users, such as
        + bank account details, credit card data, healthcare data, session tokens, social security numbers, home address, phone numbers, dates of birth, and user account information such as usernames and passwords,
      * Sensitive commercial or business data
      * Technical details about the website and its infrastructure
    - **Some basic examples of information disclosure are as follows:**
      * Revealing the names of hidden directories, their structure, and their contents via a robots.txt file or directory listing
      * Providing access to source code files via temporary backups
      * Explicitly mentioning database table or column names in error messages
      * Unnecessarily exposing highly sensitive information, such as credit card details
      * Hard-coding API keys, IP addresses, database credentials, and so on in the source code
      * Hinting at the existence or absence of resources, usernames, and so on via subtle differences in application behavior
  + **information disclosure Impact**
    - It can have both direct and indirect impact depending on the exposed data,
    - For example, leaking customers' credit card details has a high impact, these types of data require protection as defined by laws such as EU GDPR or local security laws.
  + **How to test for information disclosure vulnerabilities**
    - The following are some examples of high-level techniques and tools that you can use to help identify information disclosure vulnerabilities during testing.
      * **Fuzzing**
        + You can use the [Logger++](https://portswigger.net/bappstore/470b7057b86f41c396a97903377f3d81) extension, available from the BApp store. In addition to logging requests and responses from all of Burp's tools, it allows you to define advanced filters for highlighting interesting entries. This is just one of the many Burp extensions that can help you find any sensitive data that is leaked by the website.
      * **Using Burp Scanner**
        + Burp Suite Professional users have the benefit of Burp Scanner. This provides live scanning features for auditing items while you browse, or you can schedule automated scans to crawl and audit the target site on your behalf. Both approaches will automatically flag many information disclosure vulnerabilities for you. For example, Burp Scanner will alert you if it finds sensitive information such as private keys, email addresses, and credit card numbers in a response. It will also identify any backup files, directory listings, and so on.
      * **Using Burp's engagement tools**
        + Burp provides several engagement tools that you can use to find interesting information in the target website more easily. You can access the engagement tools from the context menu - just right-click on any HTTP message, Burp Proxy entry, or item in the site map and go to "Engagement tools". The following tools are particularly useful in this context.
        + **Search**

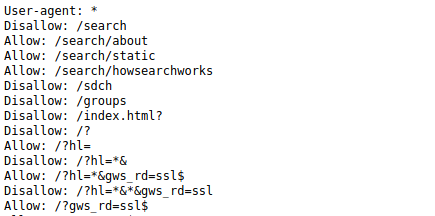
You can use this tool to look for any expression within the selected item. You can fine-tune the results using various advanced search options, such as regex search or negative search. This is useful for quickly finding occurrences (or absences) of specific keywords of interest.

* + - * + **Find comments**

You can use this tool to quickly extract any developer comments found in the selected item. It also provides tabs to instantly access the HTTP request/response cycle in which each comment was found.

* + - * + **Discover content**

You can use this tool to identify additional content and functionality that is not linked from the website's visible content. This can be useful for finding additional directories and files that won't necessarily appear in the site map automatically.

* + - * **Engineering informative responses**
        + Verbose error messages can sometimes disclose interesting information while you go about your normal testing workflow. However, by studying the way error messages change according to your input, you can take this one step further. In some cases, you will be able to manipulate the website in order to extract arbitrary data via an error message.
        + There are numerous methods for doing this depending on the particular scenario you encounter. One common example is to make the application logic attempt an invalid action on a specific item of data. For example, submitting an invalid parameter value might lead to a stack trace or debug response that contains interesting details. You can sometimes cause error messages to disclose the value of your desired data in the response.
  + **Data Leaks What Information Matters?** 
    - There are a few categories of data that have instant and recognizable value. It should be clear to just about any developer that these should be treated as higher value pieces of information in any threat-modeling exercise.
    - **API Keys** 
      * API keys are typically used to provide project-level authorization for an API, service, or other organization-type object. APIs can be critical pieces of information to expose because of the extent of their permissions and the generally wider scope of API keys.
      * A ready example of an API key might be the API key for a SaaS app, such as Twilio. A Twilio API Key doesn't differentiate access based on the role of the user; it just gives everyone who has it the ability to make API calls to the associated Twilio account.
    - **Access Tokens** 
      * Tokens are different from API keys. Access tokens are usually used to authenticate an individual (for example, session tokens and generally all cookies) as opposed to an entire service or project. Access tokens can still be sensitive data, depending on the scope of the token's authentication. API keys are something that should generally never be public (unless it's the public half of a multi-key system) but your browser trades session authentication tokens back and forth with the sites you visit every day. These distinctions aren't ironclad they only describe a convention that can be freely broken but they do provide a great jumping-off point for understanding some of the distinctions between different kinds of authentication data.
      * A common example of a popular access token would be an AWS Identity and Access Management (IAM) access token, which provides the basis for regulating an IAM role's access to different Amazon resources owned by the larger organizational account.
    - **Passwords** 
      * This is a no-brainer. Team/role-based and individual passwords, if stored in plaintext (or insufficiently encrypted) and exposed, are obviously dangerous points of vulnerability that hackers can use to infiltrate even more privileged systems. The username/password credential pattern underpins most of the services consumers interact with regularly, from social media profiles to bank accounts.
    - **Hostnames** 
      * This can be a bit more of a gray area. Quite often, if a hostname is exposed in publicly available logs or in an API, if it's meant to be internal, it will be locked down to a VPN or privileged network. However, if they aren't protected by a VPN or firewall, even the IP or hostname of a box can be an exploitable liability.
    - **Machine RSA/Encryption Keys** 
      * Unlike API keys, which describe permissions for services, projects, and roles, a machine RSA, or similar key, represents the cryptographic identity of an individual machine (whether it's a laptop, server, and so on). Exposed RSA keys for even lesser services, such as continuous deployment build servers for smaller or staging environments, can provide the necessary foothold for an attacker to inject malicious elements into other parts of your network. If you're using a macOS-powered machine, you'll typically store the SSH keys associated with your machine in a hidden .ssh folder. A typical naming convention is id\_rsa for your private key and id\_rsa.pub for your public one.
    - **Account and Application Data** 
      * The information we've described up until now has all existed at the network level, with the exception of access tokens tied to in-app behavior (like session cookies). But data within the account itself, account settings, billing information, application configs, and so on are all valuable targets for any attacker.
  + **Data Leaks – Low Data Values**
    - **Generally Descriptive Error Messages** 
      * Although error messages can be a valid source of sensitive information that's only if, well, the message contains sensitive data. By itself, a stack trace that includes function names, exception types, and other common debugging info is not a vulnerability. The key differentiator here is: can you imagine an attack scenario using the information?
    - **404 and Other Non-200 Error Codes** 
      * 404s and more exotic error codes are part of the normal functioning of an application. If sensitive information is exposed in a message, that's an issue, but otherwise, the code is to be expected.
    - **Username Enumeration** 
      * Savvy sites will contain error messages for sign-up and login pages that don't indicate whether a username exists: invalid credentials are vague enough to make it unclear whether it was the username or password that was incorrect, while the message username already exists instantly tells an attacker that there's a valid user target with that account. Combined with a script that fuzzes different possible usernames (based on something like a dictionary attack), a determined assailant could create a list of all the site's users. Regardless, because it's so resource-intensive, common, and since it doesn't lead directly to a serious vulnerability like remote code execution, username enumeration does not merit a bug bounty payout for most companies.
    - **Browser Autocomplete or Save Password Functionality**
      * Enabling a browser's form autocomplete or save password functionality is often recommended against because attackers who gain access to your browser can look back to leverage stored credentials. Since it already depends on another vulnerability to allow an attacker to access your browser in the first place, this bug does not merit a bounty payout.
    - **Data Leak Vectors**
      * So far, we've listed different types of information, but not where we can expect to find anything. Here are a few places where a website or app can unintentionally expose sensitive information.
      * **Config Files** 
        + Config management is an entire branch of operations that ensures configuration credentials are never exposed. Whether you're injecting them at runtime via a service such as consul or simply leaving them unversioned by including them in your project’s .gitignore, there are varying degrees of sophistication in the available solutions. But sometimes those measures fail and a config file is included in a server's root directory, logs on an exposed build server, application error messages, or a public code repository. That can make the sensitive contents of that config fair game for any attackers. Earlier, we discussed discovering sensitive config files in the context of applying fuzzing tools such as wfuzz that use wordlists to attempt to access files that have been left on a web server and mistakenly left accessible. We used the seclists repository of curated pentesting resources
      * **Public Code Repos** 
        + With more developers using open-source sites, such as GitHub, to network and share code, it's easy for flat file credentials and text-based secrets to be mistakenly included in a repo's commit history. It's important to note here that if you mistakenly commit sensitive data to your project's Git history, the first thing you should do is rotate those credentials. Don't try and push a commit removing the info (keep in mind, it can still be found in a previous commit); just refresh those API keys or passwords first, and then worry about removing the info from the repo later.
* **COMMON Sources of Source code disclosure**
  + **Information disclosure through Developers comments**
    - Sometimes the developer forgets to remove important comments in the web applications (e.g. default access creds) before pushing it to production
    - the source code can sometimes reveal some information disclosure
  + **Files for web crawlers (robots.txt)**
    - Many websites provide files at /robots.txt and /sitemap.xml to help crawlers navigate their site. Among other things, these files often list specific directories that the crawlers should skip, for example, because they may contain sensitive information.
    - robots.txt tells search engine crawlers which pages or files the crawler can or can't request from your site.
      * 
    - As these files are not usually linked from within the website, they may not immediately appear in Burp's site map. However, it is worth trying to navigate to /robots.txt or /sitemap.xml manually to see if you find anything of use.
  + **File Name & File Path Disclosure**
    - A web application may disclose the structure of underlying infrastructure by revealing either file names or file paths or both. Due to inappropriate input handling, improper configuration management, or backend exceptions, a web application’s response may include such information in error pages.
  + **Information Disclosure Through Directory Listing**
    - Related to filename and path disclosure is directory listing in web servers. This functionality is provided by default on web servers. When there is no default web page to show the web server shows the user a list of files and directories present on the website.
    - Web servers can be configured to automatically list the contents of directories that do not have an index page present. This can aid an attacker by enabling them to quickly identify the resources at a given path, and proceed directly to analyzing and attacking those resources. It particularly increases the exposure of sensitive files within the directory that are not intended to be accessible to users, such as temporary files and crash dumps.
      * Therefore, if the default filename on an Apache web server is *index.php*, and you have not uploaded a file called *index.php* in the root directory of your website, the server will show a directory listing of the root directory instead of parsing the php file
    - Directory listings themselves are not necessarily a security vulnerability. However, if the website also fails to implement proper access control, leaking the existence and location of sensitive resources in this way is clearly an issue.
  + **Brute forcing web application directories/files**
    - There are some useful tools that can be used to guess the hidden directories/files in the web application, like dirb, dirsearch, dirbuster, etc.
    - fuzz.txt
      * //world list that contains a lot of directory names we can use it with dirb or any tool (by booM in git)
  + **Information Disclosure Through ERROR MESSAGES**
    - One of the most common causes of information disclosure is verbose error messages. As a general rule, you should pay close attention to all error messages you encounter during auditing.
    - The content of error messages can reveal information about what input or data type is expected from a given parameter. This can help you to narrow down your attack by identifying exploitable parameters. It may even just prevent you from wasting time trying to inject payloads that simply won't work.
    - Verbose error messages can also provide information about different technologies being used by the website. For example, they might explicitly name a template engine, database type, or server that the website is using, along with its version number. This information can be useful because you can easily search for any documented exploits that may exist for this version. Similarly, you can check whether there are any common configuration errors or dangerous default settings that you may be able to exploit. Some of these may be highlighted in the official documentation.
    - You might also discover that the website is using some kind of open-source framework. In this case, you can study the publicly available source code, which is an invaluable resource for constructing your own exploits.
    - Differences between error messages can also reveal different application behavior that is occurring behind the scenes. Observing differences in error messages is a crucial aspect of many techniques, such as [SQL injection](https://portswigger.net/web-security/sql-injection), [username enumeration](https://portswigger.net/web-security/authentication/password-based#username-enumeration), and so on.
    - **Error messages Examples:**
      * Script Error Messages
      * stack traces: is debugging information about the error which also includes information about the path of files and often the piece of code where the error originated.
      * Informative Debug Message
      * Server and Database Message: they often disclose the query that generated the error, enabling you to ﬁne-tune a SQL injection attack
    - **Note**
      * we can hide our server info from banner grabbing and use generic error messages
        + ex: In Apache, custom error pages can be conﬁgured using the ErrorDocument directive in httpd.conf:

ErrorDocument 500 /generalerror.html

* + **Source code disclosure via backup files (.old/.bak/.php~)**
    - Occasionally, it is even possible to cause the website to expose its own source code. When mapping out a website, you might find that some source code files are referenced explicitly. Unfortunately, requesting them does not usually reveal the code itself. When a server handles files with a particular extension, such as .php, it will typically execute the code, rather than simply sending it to the client as text. However, in some situations, you can trick a website into returning the contents of the file instead. For example, text editors often generate temporary backup files while the original file is being edited. These temporary files are usually indicated in some way, such as by appending a tilde (~) to the filename or adding a different file extension. Requesting a code file using a backup file extension can sometimes allow you to read the contents of the file in the response.
    - The issue occurs when the code of the backend environment of a web application is exposed to the public such as an admin made a backup of the database or source code and put it on the server or the site so anyone can download it its usually compressed (zip/tar/sql/tgz/...)
    - **hidden files (.filename)** 
      * **ex of hidden files source code repositories (.git , .svn) (git repository)**
    - **backup files extensions**
      * txt,log = log files with text
      * admin.php~ = hidden copies of editable files
      * .zip , .gz , .tar , tar.gz, .rar , .bz2 , .7z , .tgz , .php.bak = archives that may contain backup files (e.g., backup.zip, backup.tar.gz)
      * conf,cnf = config files (e.g., admin.conf)
      * .sql , .bak , .old, .json= sql database backups
      * .git = version control system files
    - **Bruteforce Example**
      * dirb http://example.com/ -X .php~,.php.bak,.txt,.log,.php,.html,.zip,.gz,.tar,.gz,.sql,.bak,.conf,.cnf,.old
      * dirsearch.py --url http://example.com/ -e php~,txt,log,php,html,zip,gz,tar,gz,sql,bak,backup,conf,cnf
      * nmap -p 80 --script=http-backup-finder --script-args http-backup-finder.url=/web-serveur/ch11/index.php challenge01.root-me.org
      * wfuzz -c -z file, /usr/share/worldlists/dirbuster/directory-list-2.3-medium.txt --hc 404  http://website/FUZZ    //fuzz directories
      * wfuzz -c -z file, /usr/share/worldlists/dirbuster/directory-list-2.3-medium.txt --hc 404  http://website/FUZZ.php   //fuzz files     {.php, .back, .zip, .sh, }
  + **Checking JavaScript files for useful URLs / endpoints**
    - JavaScript files (a lot of JS files contains important info)
    - Sometimes you could find useful endpoints in the JS files that are being used by the web application, there are a lot of tools that can do that automatically like:
      * LinkFinder , this tool will bring all hidden endpoints and their parameters to you .
  + **Information disclosure through git directories / GitHub repository** 
    - **Git directories** 
      * Sometimes you will find .git directory exposed in the website, this can lead you to get sensitive information , or even reading the source code of that web application .
      * Virtually all websites are developed using some form of version control system, such as Git/SVN. By default, a Git project stores all of its version control data in a folder called .git. Occasionally, websites expose this directory in the production environment. In this case, you might be able to access it by simply browsing to /.git.
      * While it is often impractical to manually browse the raw file structure and contents, there are various methods for downloading the entire .git directory. You can then open it using your local installation of Git to gain access to the website's version control history. This may include logs containing committed changes and other interesting information.
      * This might not give you access to the full source code, but comparing the diff will allow you to read small snippets of code. As with any source code, you might also find sensitive data hard-coded within some of the changed lines.
        + wget -r http://website/.git
        + git status
        + git diff
        + diff hash\_file1 hash\_file2

diff 2472e424e9fc96a77572f72664a8561c17d1da 1eed2d0dd59dc238b9726005bb2ff6b2070ca6

* + - * + git checkout HEAD <filename>

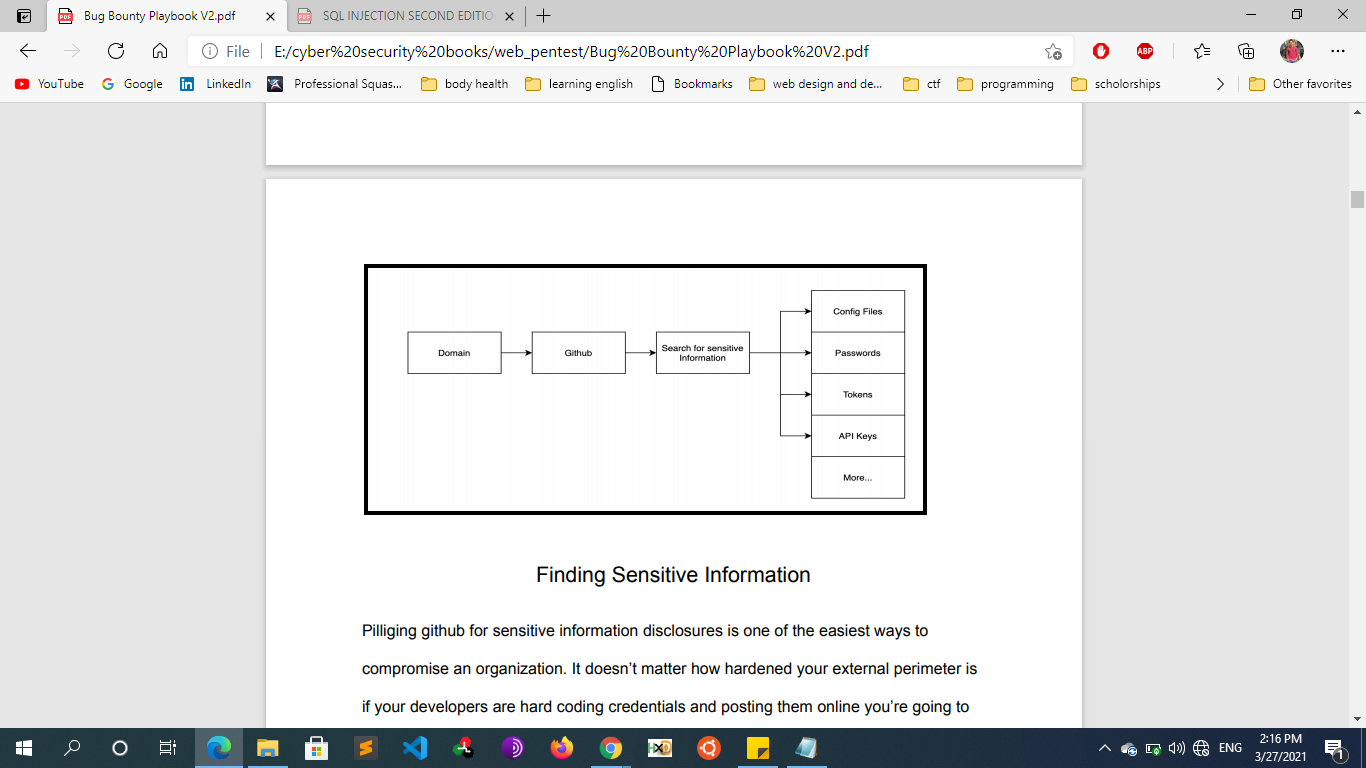
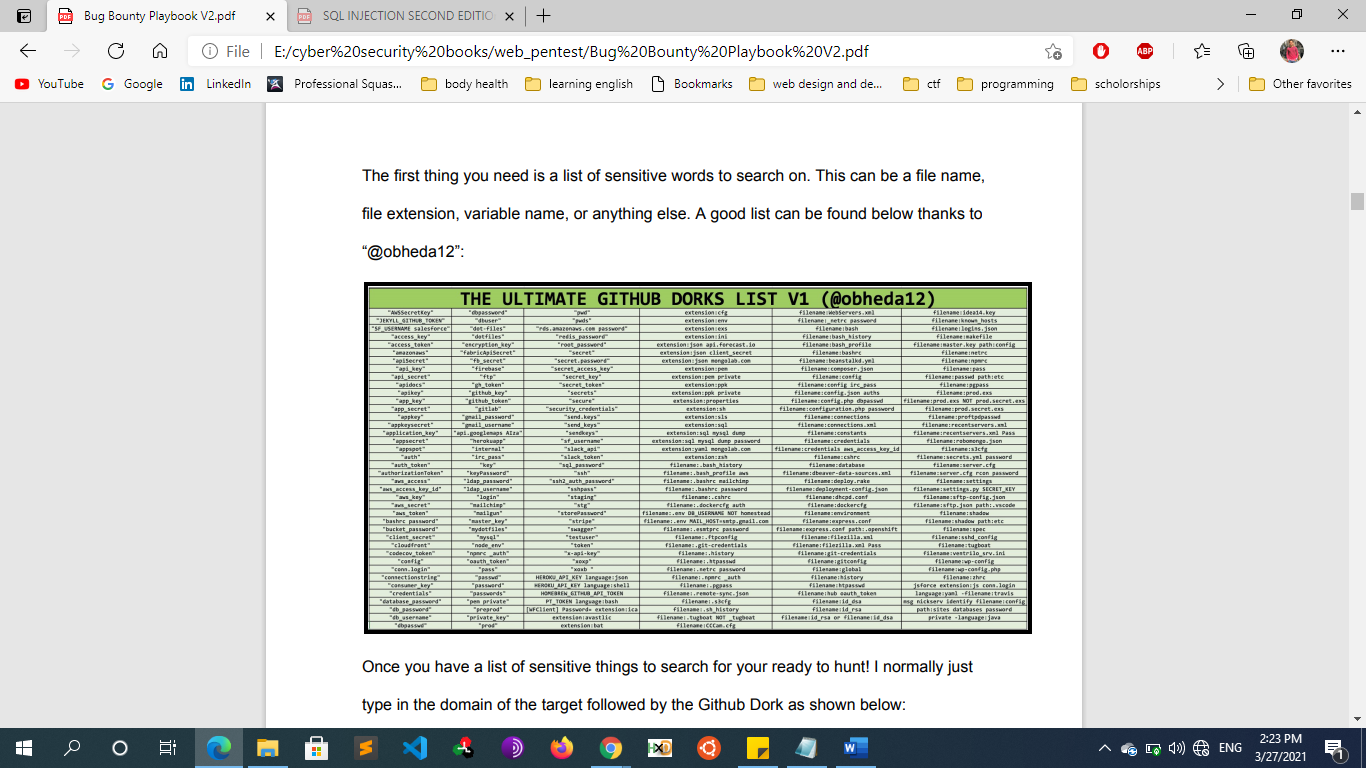
revert a deleted/updated file

* + - * + git show: will show you the last commit changes content
        + git log: give you the history of current branch, it shows you all branch commits
        + git reset: will reset the branch to the specific commit. and we can run git show to get all changes inside it .

git reset 1eed2d0dd59dc238b9726005bb2ff6b2070ca69e

* + - * **Notes** 
        + **what is dvcs pillage**

(Dump any website that has git/svn repository on the server)

* + - **Searching in GitHub repository**
      * Also, some organization uses public GitHub repositories in which you can look for any hidden secrets (e.g. API keys, etc.), so search in GitHub for the website maybe we will find something. you can use some tools for that like:
        + [**truffleHog**](https://github.com/dxa4481/truffleHog) : checks git repositories for secret information
        + [**Gitrob**](https://github.com/michenriksen/gitrob) : reconnaissance tool for GitHub organizations
        + 
      * Pillaging GitHub for sensitive information disclosures is one of the easiest ways to compromise an organization. It doesn’t matter how hardened your external perimeter is if your developers are hard coding credentials and posting them online, you’re going to get compromised. It's fairly common for developers to hard code test accounts, API keys, or whatever when they are writing a piece of software. This makes things easy for the developer as they won’t have to enter their credentials every time they go to run/test their program. However, more times than not these credentials remain in the source code when they push it to GitHub, if this repository is public everyone can view it.
      * The first thing you need is a list of sensitive words to search on. This can be a file name, file extension, variable name, or anything else. A good list can be found below
      * 
      * Once you have a list of sensitive things to search for your ready to hunt! I normally just type in the domain of the target followed by the GitHub Dork as shown below:
        + Domain.com “password”
      * **Example:** 
        + searching for the domain “hackerone.com” and the term “password” gave us 7,390 results. In a typical scenario I would end up going through 90% of these results by hand for a few hours before I find something juicy. Having to spend hours sorting through a bunch of trash is really the only downside to this technique. However, when you do find something, it typically leads to a high or critical finding.
  + **Information disclosure due to insecure configuration**
    - Websites are sometimes vulnerable as a result of improper configuration. This is especially common due to the widespread use of third-party technologies, whose vast array of configuration options are not necessarily well-understood by those implementing them.
    - In other cases, developers might forget to disable various debugging options in the production environment. For example, the HTTP TRACE method is designed for diagnostic purposes. If enabled, the web server will respond to requests that use the TRACE method by echoing in the response the exact request that was received. This behavior is often harmless, but occasionally leads to information disclosure, such as the name of internal authentication headers that may be appended to requests by reverse proxies.
* **Preventing Data Exposure**
  + Classify data processed, stored, or transmitted by an application. And identify which data is sensitive according to privacy laws, regulatory requirements, or business needs.
  + Apply controls as per the classification.
  + Discard it as soon as possible or use PCI DSS compliant tokenization or even truncation. Data that is not retained cannot be stolen.
  + Make sure to encrypt all sensitive data and Store passwords using strong adaptive and salted hashing functions with a work factor (delay factor), such as Argon2, scrypt, bcrypt, or PBKDF2.
  + Ensure up-to-date and strong standard algorithms, protocols, and keys are in place; use proper key management.
  + Encrypt all data in transit with secure protocols such as TLS with perfect forward secrecy (PFS) ciphers, cipher prioritization by the server, and secure parameters.
  + Enforce encryption using directives like HTTP Strict Transport Security (HSTS).
  + Disable caching for responses that contain sensitive data.
  + Verify independently the effectiveness of configuration and settings.
  + Make sure that everyone involved in producing the website is fully aware of what information is considered sensitive. Sometimes seemingly harmless information can be much more useful to an attacker than people realize. Highlighting these dangers can help make sure that sensitive information is handled more securely in general by your organization.
  + Audit any code for potential information disclosure as part of your QA or build processes. It should be relatively easy to automate some of the associated tasks, such as stripping developer comments.
  + Use generic error messages as much as possible. Don't provide attackers with clues about application behavior unnecessarily.
  + Double-check that any debugging or diagnostic features are disabled in the production environment.
  + Make sure you fully understand the configuration settings, and security implications, of any third-party technology that you implement. Take the time to investigate and disable any features and settings that you don't actually need.
* **Resources** 
  + https://portswigger.net/web-security/information-disclosure/exploiting
* **Cross Site Script Inclusion (XSSI)**
  + **Overview**
    - Cross Site Script Inclusion (XSSI) is an attack technique (or a vulnerability) that enables attackers to steal data of certain types across origin boundaries, by including target data using SCRIPT tag in an attacker's Web page:
      * <!-- attacker's page loads external data with SCRIPT tag -->
      * <SCRIPT src="http://target.example.jp/secret"></SCRIPT>
    - exploitation of a Cross-Site Scripting Include (XSSI) vulnerability. This type of vulnerability is common with modern applications and applications relying massively on JSON with Padding (JSONP).
    - exploits the fact that, when a resource is included using the script tag, the SOP doesn’t apply, because scripts have to be able to be included cross-domain. An attacker can thus read everything that was included using the script tag.
    - The main issue with this application is that sensitive data is exposed when a user is accessing a JavaScript page. Since the application relies on cookies for the authentication and that nothing prevents a malicious server from requesting the same JavaScript page, it's possible to get access to the sensitive information by building a very simple malicious page).
    - Browsers prevent pages of one domain from reading pages in other domains. But they do not prevent pages of a domain from referencing resources in other domains. In particular, they allow images to be rendered from other domains and scripts to be executed from other domains. An included script doesn't have its own security context. It runs in the security context of the page that included it. For example, if www.evil.example.com includes a script hosted on www.google.com then that script runs in the evil context not in the google context. So any user data in that script will "leak."
  + **XSSI vs XSS and CSRF** 
    - XSSI are by name close to Cross-Site Scripting (XSS) and from the description close to Cross-Site Request Forgery (CSRF). The commonality between the three is that they are all client-side attacks.
    - The difference to XSS is easy: during an XSS malicious code is placed into a victim’s page, during an XSSI victim’s code is included in a malicious page. On the surface XSSI and CSRF look similar, because in both a request is sent from a malicious page to a different domain and in both cases the request is executed in the context of the logged in user. The key difference is the goal. In a CSRF, the attacker wants to execute state-changing actions inside a victim’s page, like transferring money in an online banking application. In an XSSI the attacker wants to leak data cross-origin, in order to then execute one of the aforementioned attacks.
  + **SEARCHING, FINDING AND EXPLOITING**
    - When searching for XSSI one needs to distinguish four situations. But exploitation is fortunately often similar or even the same (analog to reflected XSS and stored XSS). We can differentiate four situations:
      * 1. Static JavaScript (regular XSSI)
      * 2. Static JavaScript, which is only accessible when authenticated
      * 3. Dynamic JavaScript
      * 4. Non-JavaScript
  + **Resources** 
    - https://www.scip.ch/en/?labs.20160414