TryHackMe John The Ripper

Saikat Karmakar | Jul 25 : 2021

John

John the Ripper is one of the most well known, well-loved and versatile hash cracking tools out there. It combines a fast cracking speed, with an extraordinary range of compatible hash types. This room will assume no previous knowledge, so we must first cover some basic terms and concepts before we move into practical hash cracking.

What are Hashes?

A hash is a way of taking a piece of data of any length and representing it in another form that is a fixed length. This masks the original value of the data. This is done by running the original data through a hashing algorithm. There are many popular hashing algorithms, such as MD4,MD5, SHA1 and NTLM. Lets try and show this with an example:

If we take "polo", a string of 4 characters- and run it through an MD5 hashing algorithm, we end up with an output of: b53759f3ce692de7aff1b5779d3964da a standard 32 character MD5 hash.

Likewise, if we take "polomints", a string of 9 characters- and run it through the same MD5 hashing algorithm, we end up with an output of: 584b6e4f4586e136bc280f27f9c64f3b another standard 32 character MD5 hash.

What makes Hashes secure?

Hashing algorithms are designed so that they only operate one way. This means that a calculated hash cannot be reversed using just the output given. This ties back to a fundamental mathematical problem known as the W P vs NP relationship .

While this is an extremely interesting mathematical concept that proves fundamental to computing and cryptography I am in no way qualified to try and explain it in detail here; but abstractly it means that the algorithm to hash the value will be "NP" and can therefore be calculated reasonably. However an un-hashing algorithm would be "P" and intractable to solve- meaning that it cannot be computed in a reasonable time using standard computers.

Where John Comes in...

Even though the algorithm itself is not feasibly reversible. That doesn't mean that cracking the hashes is impossible. If you have the hashed version of a password, for example- and you know the hashing algorithm- you can use that hashing algorithm to hash a

large number of words, called a dictionary. You can then compare these hashes to the one you're trying to crack, to see if any of them match. If they do, you now know what word corresponds to that hash- you've cracked it!

This process is called a **dictionary attack** and John the Ripper, or John as it's commonly shortened to, is a tool to allow you to conduct fast brute force attacks on a large array of different hash types.

Setting Up John The Ripper

John the Ripper is supported on many different Operating Systems, not just Linux Distributions. As a note before we go through this, there are multiple versions of John, the standard "core" distribution, as well as multiple community editions- which extend the feature set of the original John distribution. The most popular of these distributions is the "Jumbo John"- which we will be using specific features of later.

Parrot, Kali and AttackBox

If you're using Parrot OS, Kali Linux or TryHackMe's own AttackBox- you should already have Jumbo John installed. You can double check this by typing `john` into the terminal. You should be met with a usage guide for john, with the first line reading: "John the Ripper 1.9.0-jumbo-1" or similar with a different version number. If not, you can use `sudo apt install john` to install it.

Blackarch

If you're using Blackarch, or the Blackarch repositories you may or may not have Jumbo John installed, to check if you do, use the command `pacman -Qe | grep "john"` You should be met with an output similar to "john 1.9.0.jumbo1-5" or similar with a different version number. If you do not have it installed, you can simply use `pacman -S john` to install it.

Building from Source for Linux

If you wish to build the package from source to meet your system requirements, you can do this in five fairly straightforward steps. Further advice on the installation process and how to configure your build from source can be found here.

1. Use `git clone https://github.com/openwall/john -b bleeding-jumbo john` to clone the jumbo john repository to your current working
2. Then `cd john/src/` to change your current directory to where the source code is.
3. Once you're in this directory, use `./configure` to check the required dependencies and options that have been configured.
4. If you're happy with this output, and have installed any required dependencies that are needed, use `make -s clean && make -sj4` to build a binary of john. This binary will be in the above run directory, which you can change to with `cd ../run`
5. You can test this binary using ./john --test

Installing on Windows

To install Jumbo John the Ripper on Windows, you just need to download and install the zipped binary for either 64 bit systems here or for 32 bit systems here.

Cracking Basic Hashes

There are multiple ways to use John the Ripper to crack simple hashes, we're going to walk through a few, before moving on to cracking some ourselves.

John Basic Syntax

The basic syntax of John the Ripper commands is as follows. We will cover the specific options and modifiers used as we use them.

`john [options] [path to file]`

`john` - Invokes the John the Ripper program

`[path to file]` - The file containing the hash you're trying to crack, if it's in the same directory you won't need to name a path, just the file.

Automatic Cracking

John has built-in features to detect what type of hash it's being given, and to select appropriate rules and formats to crack it for you, this isn't always the best idea as it can be unreliable- but if you can't identify what hash type you're working with and just want to try cracking it, it can be a good option! To do this we use the following syntax:

`john --wordlist=[path to wordlist] [path to file]`

`--wordlist=` - Specifies using wordlist mode, reading from the file that you supply in the following path...

`[path to wordlist]` - The path to the wordlist you're using, as described in the previous task.

Example Usage:

john --wordlist=/usr/share/wordlists/rockyou.txt hash_to_crack.txt

Identifying Hashes

Sometimes John won't play nicely with automatically recognising and loading hashes, that's okay! We're able to use other tools to identify the hash, and then set john to use a specific format. There are multiple ways to do this, such as using an online hash identifier like this one. I like to use a tool called hash-identifier, a Python tool that is super easy to use and will tell you what different types of hashes the one you enter is likely to be, giving you more options if the first one fails.

To use hash-identifier, you can just pull the python file from gitlab using: `wget https://gitlab.com/kalilinux/packages/hash-identifier/-/raw/kali/master/hash-id.py`.

Then simply launch it with `python3 hash-id.py` and then enter the hash you're trying to identify- and it will give you possible formats!

Format-Specific Cracking

Once you have identified the hash that you're dealing with, you can tell john to use it while cracking the provided hash using the following syntax:

`john --format=[format] --wordlist=[path to wordlist] [path to file]`

`--format=` - This is the flag to tell John that you're giving it a hash of a specific format, and to use the following format to crack it

`[format]` - The format that the hash is in

Example Usage:

```
john --format=raw-md5 --wordlist=/usr/share/wordlists/rockyou.txt hash_to_crack.txt
```

A Note on Formats:

When you are telling john to use formats, if you're dealing with a standard hash type, e.g. md5 as in the example above, you have to prefix it with `raw-` to tell john you're just dealing with a standard hash type, though this doesn't always apply. To check if you need to add the prefix or not, you can list all of John's formats using `john --list=formats` and either check manually, or grep for your hash type using something like `john --list=formats | grep -iF "md5"`.

Cracking Windows Hashes

Now that we understand the basic syntax and usage of John the Ripper- lets move on to cracking something a little bit more difficult, something that you may even want to attempt if you're on a real Penetration Test or Red Team engagement. Authentication hashes are the hashed versions of passwords that are stored by operating systems, it is sometimes possible to crack them using the brute-force methods that we're using. To get your hands on these hashes, you must often already be a privileged user- so we will explain some of the hashes that we plan on cracking as we attempt them.

NTHash / NTLM

NThash is the hash format that modern Windows Operating System machines will store user and service passwords in. It's also commonly referred to as "NTLM" which references the previous version of Windows format for hashing passwords known as "LM", thus "NT/LM".

A little bit of history, the NT designation for Windows products originally meant "New Technology", and was used- starting with W Windows NT, to denote products that were not built up from the MS-DOS Operating System. Eventually, the "NT" line became the standard Operating System type to be released by Microsoft and the name was dropped, but it still lives on in the names of some Microsoft technologies.

You can acquire NTHash/NTLM hashes by dumping the SAM database on a Windows machine, by using a tool like Mimikatz or from the Active Directory database: NTDS.dit. You may not have to crack the hash to continue privilege escalation- as you can often conduct a "pass the hash" attack instead, but sometimes hash cracking is a viable option if there is a weak password policy.

Cracking Hashes from /etc/shadow

The /etc/shadow file is the file on Linux machines where password hashes are stored. It also stores other information, such as the date of last password change and password expiration information. It contains one entry per line for each user or user account of the system. This file is usually only accessible by the root user- so in order to get your hands on the hashes you must have sufficient privileges, but if you do- there is a chance that you will be able to crack some of the hashes.

Unshadowing

John can be very particular about the formats it needs data in to be able to work with it, for this reason- in order to crack /etc/shadow passwords, you must combine it with the /etc/passwd file in order for John to understand the data it's being given. To do this, we use a tool built into the John suite of tools called unshadow. The basic syntax of unshadow is as follows:

`unshadow [path to passwd] [path to shadow]`

`unshadow` - Invokes the unshadow tool

`[path to passwd]` - The file that contains the copy of the /etc/passwd file you've taken from the target machine

`[path to shadow]` - The file that contains the copy of the /etc/shadow file you've taken from the target machine

Example Usage:

unshadow local_passwd local_shadow > unshadowed.txt

Note on the files

When using unshadow, you can either use the entire /etc/passwd and /etc/shadow file- if you have them available, or you can use the relevant line from each, for example:

FILE 1 - local_passwd

Contains the /etc/passwd line for the root user:

root:x:0:0::/root:/bin/bash

FILE 2 - local_shadow

Contains the /etc/shadow line for the root user:

root:$6$2nwjN454g.dv4HN/m9Z/r2xVfweYVkrr.v5Ft8Ws3/YYksfNwq96UL1FX0OJjY1L6l.DS3KEVsZ9rOVLB/ldTeEL/OIhJZ4GMFMGA0:18576::::::

Cracking

We're then able to feed the output from unshadow, in our example use case called "unshadowed.txt" directly into John. We should not need to specify a mode here as we have made the input specifically for John, however in some cases you will need to specify the format as we have done previously using: `--format=sha512crypt`

`john --wordlist=/usr/share/wordlists/rockyou.txt --format=sha512crypt unshadowed.txt`

Single Crack Mode

So far we've been using John's wordlist mode to deal with brute forcing simple., and not so simple hashes. But John also has another mode, called Single Crack mode. In this mode, John uses only the information provided in the username, to try and work out possible passwords heuristically, by slightly changing the letters and numbers contained within the username.

Word Mangling

The best way to show what Single Crack mode is, and what word mangling is, is to actually go through an example:

If we take the username: Markus

Some possible passwords could be:

- Markus1, Markus2, Markus3 (etc.)
- MArkus, MARkus, MARKus (etc.)

- Markus!, Markus$, Markus* (etc.)

This technique is called word mangling. John is building it's own dictionary based on the information that it has been fed and uses a set of rules called "mangling rules" which define how it can mutate the word it started with to generate a wordlist based off of relevant factors for the target you're trying to crack. This is exploiting how poor passwords can be based off of information about the username, or the service they're logging into.

GECOS

John's implementation of word mangling also features compatibility with the Gecos fields of the UNIX operating system, and other UNIX-like operating systems such as Linux. So what are Gecos? Remember in the last task where we were looking at the entries of both /etc/shadow and /etc/passwd? Well if you look closely You can see that each field is seperated by a colon ":". Each one of the fields that these records are split into are called Gecos fields. John can take information stored in those records, such as full name and home directory name to add in to the wordlist it generates when cracking /etc/shadow hashes with single crack mode.

Using Single Crack Mode

To use single crack mode, we use roughly the same syntax that we've used to so far, for example if we wanted to crack the password of the user named "Mike", using single mode, we'd use:

```
john --single --format=[format] [path to file]
```

`--single` - This flag lets john know you want to use the single hash cracking mode.

Example Usage:

```
john --single --format=raw-sha256 hashes.txt
```

A Note on File Formats in Single Crack Mode:

If you're cracking hashes in single crack mode, you need to change the file format that you're feeding john for it to understand what data to create a wordlist from. You do this by prepending the hash with the username that the hash belongs to, so according to the above example- we would change the file hashes.txt

From:

```
1efee03cdcb96d90ad48ccc7b8666033
```

To

```
mike:1efee03cdcb96d90ad48ccc7b8666033
```

Custom Rules

What are Custom Rules?

As we journeyed through our exploration of what John can do in Single Crack Mode- you may have some ideas about what some good mangling patterns would be, or what patterns your passwords often use- that could be replicated with a certain mangling pattern. The good news is you can define your own sets of rules, which John will use to dynamically create passwords. This is especially useful when you know more information about the password structure of whatever your target is.

Common Custom Rules

Many organisations will require a certain level of password complexity to try and combat dictionary attacks, meaning that if you create an account somewhere, go to create a password and enter:

polopassword

You may receive a prompt telling you that passwords have to contain at least one of the following:

- Capital letter
- Number
- Symbol

This is good! However, we can exploit the fact that most users will be predictable in the location of these symbols. For the above criteria, many users will use something like the following:

Polopassword1!

A password with the capital letter first, and a number followed by a symbol at the end. This pattern of the familiar password, appended and prepended by modifiers (such as the capital letter or symbols) is a memorable pattern that people will use, and reuse when they create passwords. This pattern can let us exploit password complexity predictability.

Now this does meet the password complexity requirements, however as an attacker we can exploit the fact we know the likely position of these added elements to create dynamic passwords from our wordlists.

How to create Custom Rules

Custom rules are defined in the `john.conf` file, usually located in `/etc/john/john.conf` if you have installed John using a package manager or built from source with `make` and in `/opt/john/john.conf` on the TryHackMe Attackbox.

Let's go over the syntax of these custom rules, using the example above as our target pattern. Note that there is a massive level of granular control that you can define in these rules, I would suggest taking a look at the wiki here in order to get a full view of the types of modifier you can use, as well as more examples of rule implementation.

The first line:

`[List.Rules:THMRules]` - Is used to define the name of your rule, this is what you will use to call your custom rule as a John argument.

We then use a regex style pattern match to define where in the word will be modified, again- we will only cover the basic and most common modifiers here:

`Az` - Takes the word and appends it with the characters you define

`A0` - Takes the word and prepends it with the characters you define

`c` - Capitalises the character positionally

These can be used in combination to define where and what in the word you want to modify.

Lastly, we then need to define what characters should be appended, prepended or otherwise included, we do this by adding character sets in square brackets `[ ]` in the order they should be used. These directly follow the modifier patterns inside of double quotes `" "`. Here are some common examples:

`[0-9]` - Will include numbers 0-9

`[0]` - Will include only the number 0

`[A-z]` - Will include both upper and lowercase

`[A-Z]` - Will include only uppercase letters

`[a-z]` - Will include only lowercase letters

`[a]` - Will include only a

`[!£$%@]` - Will include the symbols !£$%@

Putting this all together, in order to generate a wordlist from the rules that would match the example password "Polopassword1!" (assuming the word polopassword was in our wordlist) we would create a rule entry that looks like this:

```
[List.Rules:PoloPassword]
```

```
cAz"[0-9] [!£$%@]"
```

In order to:

Capitalise the first letter - `c`

Append to the end of the word - `Az`

A number in the range 0-9 - `[0-9]`

Followed by a symbol that is one of `[!£$%@]`

Using Custom Rules

We could then call this custom rule as a John argument using the `--rule=PoloPassword` flag.

As a full command: `john --wordlist=[path to wordlist] --rule=PoloPassword [path to file]`

As a note I find it helpful to talk out the patterns if you're writing a rule- as shown above, the same applies to writing RegEx patterns too.

Jumbo John already comes with a large list of custom rules, which contain modifiers for use almost all cases. If you get stuck, try looking at those rules [around line 678] if your syntax isn't working properly.

Cracking a Password Protected Zip File

Yes! You read that right. We can use John to crack the password on password protected Zip files. Again, we're going to be using a separate part of the john suite of tools to convert the zip file into a format that John will understand, but for all intents and purposes, we're going to be using the syntax that you're already pretty familiar with by now.

Zip2John

Similarly to the unshadow tool that we used previously, we're going to be using the zip2john tool to convert the zip file into a hash format that John is able to understand, and hopefully crack. The basic usage is like this:

```
zip2john [options] [zip file] > [output file]
```

`[options]` - Allows you to pass specific checksum options to zip2john, this shouldn't often be necessary

`[zip file]` - The path to the zip file you wish to get the hash of

`>` - This is the output director, we're using this to send the output from this file to the...

`[output file]` - This is the file that will store the output from

Example Usage

```
zip2john zipfile.zip > zip_hash.txt
```

Cracking

We're then able to take the file we output from zip2john in our example use case called "zip_hash.txt" and, as we did with unshadow, feed it directly into John as we have made the input specifically for it.

```
john --wordlist=/usr/share/wordlists/rockyou.txt zip_hash.txt
```

Cracking a Password Protected RAR Archive

We can use a similar process to the one we used in the last task to obtain the password for rar archives. If you aren't familiar, rar archives are compressed files created by the Winrar archive manager. Just like zip files they compress a wide variety of folders and files.

Rar2John

Almost identical to the zip2john tool that we just used, we're going to use the rar2john tool to convert the rar file into a hash format that John is able to understand. The basic syntax is as follows:

```
rar2john [rar file] > [output file]
```

`rar2john` - Invokes the rar2john tool

`[rar file]` - The path to the rar file you wish to get the hash of

`>` - This is the output director, we're using this to send the output from this file to the...

`[output file]` - This is the file that will store the output from

Example Usage

```
rar2john rarfile.rar > rar_hash.txt
```

Cracking

Once again, we're then able to take the file we output from rar2john in our example use case called "rar_hash.txt" and, as we did with zip2john we can feed it directly into John..

```
john --wordlist=/usr/share/wordlists/rockyou.txt rar_hash.txt
```

Cracking SSH Key Passwords

Okay, okay I hear you, no more file archives! Fine! Let's explore one more use of John that comes up semi-frequently in CTF challenges. Using John to crack the SSH private key password of id_rsa files. Unless configured otherwise, you authenticate your SSH login using a password. However, you can configure key-based authentication, which lets you use your private key, id_rsa, as an authentication key to login to a remote machine over SSH. However, doing so will often require a password- here we will be using John to crack this password to allow authentication over SSH using the key.

SSH2John

Who could have guessed it, another conversion tool? Well, that's what working with John is all about. As the name suggests ssh2john converts the id_rsa private key that you use to login to the SSH session into hash format that john can work with. Jokes aside, it's another beautiful example of John's versatility. The syntax is about what you'd expect. Note that if you don't have ssh2john installed, you can use ssh2john.py, which is located in the /opt/john/ssh2john.py. If you're doing this, replace the `ssh2john` command with `python3 /opt/ssh2john.py` or on Kali, `python /usr/share/john/ssh2john.py`.

```
ssh2john [id_rsa private key file] > [output file]
```

ssh2john - Invokes the ssh2john tool

`[id_rsa private key file]` - The path to the id_rsa file you wish to get the hash of

`>` - This is the output director, we're using this to send the output from this file to the...

`[output file]` - This is the file that will store the output from

Example Usage

ssh2john id_rsa > id_rsa_hash.txt

Cracking

For the final time, we're feeding the file we output from ssh2john, which in our example use case is called "id_rsa_hash.txt" and, as we did with rar2john we can use this seamlessly with John:

```
john --wordlist=/usr/share/wordlists/rockyou.txt id_rsa_hash.txt
```