The Predictable Password

How a wordlist is built, how big it gets, and why a life story can be a password

Dr. Ahmet Furkan Aydogan | aydogana@uncw.edu
Department of Computer Science, University of North Carolina Wilmington

Track	Fundamentals of Cyber Security (CYBR 201)
Term	Fall 2026
Instructor	Dr. Ahmet Furkan Aydogan
Deliverable	The completed Word worksheet (see the last section)
Points	100

Abstract. *A password is only a secret if it is hard to guess. In this lab you learn to build a **wordlist**, a plain file of password candidates, on your own Parrot OS machine, and you learn to count before you generate: how many candidates a character set and a length produce, and how large the file becomes on disk. You start with every six digit PIN, watch the file reach seven megabytes, and then calculate, without generating them, how the numbers explode once letters and symbols join in. Then you turn the idea around. Instead of trying every combination, you read a person's life, a fictional one, and turn their dog, their town, and their birth year into a short, targeted list that guesses their five character password in seconds. Part I explains what a wordlist is, the difference between a brute force and a dictionary attack, and how to estimate a wordlist's size before you build it. Part II is the graded exercise you carry out from your own Parrot machine, ending with one authorized login you recover yourself.*

1. PART I. What you need to understand first

The idea in one line

A locked door with a number pad falls in two ways. You can try every possible code from 0000 upward until one opens it, or you can try the codes most people actually pick: a birth year, a house number, 1234. The first way is a **brute force** attack; the second is a **dictionary** (or wordlist) attack. This lab is about building the list you try, measuring how big it gets, and learning why a password drawn from a person's own life is the easiest kind to guess.

Overview and learning objectives

By the end of this lab you will be able to:

(a) say what a wordlist is, and explain the difference between a brute force attack and a dictionary attack;

(b) calculate how many candidates a given character set and password length produce, and estimate the wordlist's file size before you build it;

(c) move around the Parrot filesystem in a terminal, build a workspace on your Desktop, and generate, inspect, and measure a wordlist;

(d) stop a long running command safely, and tell `Ctrl+C`, `Ctrl+Z`, and `Ctrl+X` apart;

(e) turn a person's biography into a small, targeted wordlist that respects a login's rules; and

(f) run controlled login attempts against your authorized lab target only, and say what defenses would have stopped you.

What a wordlist is

A **wordlist** is nothing more than a text file with one password candidate on each line. A cracking tool reads it top to bottom and tries each line against a lock, stopping the instant one fits. The famous `rockyou.txt` that ships with Parrot is exactly this: about fourteen million real passwords, one per line, leaked in a 2009 breach and kept for training. You will not need it here. You will build small lists of your own, so you can see every candidate and count them exactly.

Brute force versus dictionary: two ways to guess

Brute force means try every possible string of a given shape: for a four digit PIN, that is 0000, 0001, 0002, all the way to 9999. It always finds the answer eventually, but "eventually" can be longer than a human lifetime once the password is long. A **dictionary attack** does not try everything; it tries a curated list of *likely* candidates: common passwords, or, as in Part II, guesses drawn from what you know about the target. It is smaller and faster, but it only wins if the real password is on the list. The whole craft is making a list that is small enough to finish and smart enough to contain the answer.

Counting before you generate

Before building any list you should be able to predict its size. Two numbers decide everything: the **character set** (how many different symbols each position may hold) and the **length** (how many positions there are).

Number of candidates	(character set size) raised to the power of (length), written n^L
Approximate file size	candidates $\times$ (length + 1), because each line is L ASCII characters plus one newline byte

So a six digit PIN uses a character set of $n = 10$ (the digits 0 through 9) and length $L = 6$. That is $10^6 = 1{,}000{,}000$ candidates. Each line is six characters plus one newline, so the file is about $1{,}000{,}000 \times 7 = 7{,}000{,}000$ bytes. You will confirm both of these on the machine in Part II.

MB versus MiB. Disk tools disagree about "mega." A **megabyte (MB)** is 1,000,000 bytes; a **mebibyte (MiB)** is 1,048,576 bytes. So 7,000,000 bytes is 7.0 MB but only about 6.68 MiB. This is why `ls` may say 7000000 while `du -h` says 6.7M: they are both right, in different units.

crunch and its fill in the blank templates

The tool you will use to generate wordlists is `crunch`. In its simplest form you give it a minimum length, a maximum length, and a character set: `crunch 6 6 0123456789` makes every six digit string. But `crunch` can also keep part of a word *fixed* and vary only the rest, using a `-t` template with placeholders. This is the key to targeted guessing: you fix what you know and vary only what you do not.

Placeholder	Stands for	Example in `-t`
`@`	any lowercase letter	`-t ace@@` makes `aceaa` to `acezz`
`,`	any uppercase letter	`-t ,ce99` makes `Ace99` to `Zce99`
`%`	any digit	`-t Ace%%` makes `Ace00` to `Ace99`
`^`	any symbol	`-t Ace%^` makes `Ace0!` and its kin

Any character in the template that is *not* a placeholder is kept exactly. So `Ace` stays `Ace`, and only the two `%` positions change. The more you fix, the fewer candidates you make.

Same skill, two chairs

From the defender's chair. Counting candidates is how a security team decides whether a password policy is strong: if a rule allows only five digit PINs, an attacker needs at most 100,000 tries, which is nothing, so the team requires longer, mixed passwords and adds rate limits. **From the attacker's chair.** The same arithmetic tells an intruder whether a target is worth attacking and which list to build. The technique is neutral; the target and the authorization are what make it right or wrong.

Attack only what you are authorized to attack. Every login attempt in this lab goes to ONE place: the training panel your instructor set up for this course, at the address given in Part II. You do not point any tool at a real website, a classmate's account, the course portal, or any system you do not own. Password guessing is legal on a target you own or are explicitly authorized to test; run it against anyone else and it is a crime. This lab is taught so you can measure password strength and recognize weak choices, and so you can defend against exactly this attack. The technique is neutral; the target and the authorization are what make it right or wrong.

2. PART II. The exercise

Where your answers go. Do NOT hand in a blank document. Open the file "The Predictable Password - Student Worksheet.docx" that came with this lab. Every step and question below has a matching answer line, calculation box, or screenshot box in that worksheet. Fill in each value as you go and paste each screenshot into its box. The completed worksheet is what you submit. Work on your own Parrot machine. Do not rush; the first time it could take a while, and that is normal.

Type ONE command at a time. Press Enter, read what comes back, then move to the next line. Each step below is separated on purpose so you can see what every command does. The screenshots are real captures from a Parrot Security machine; your own output will look almost identical.

Section A. Open a terminal and build your workspace

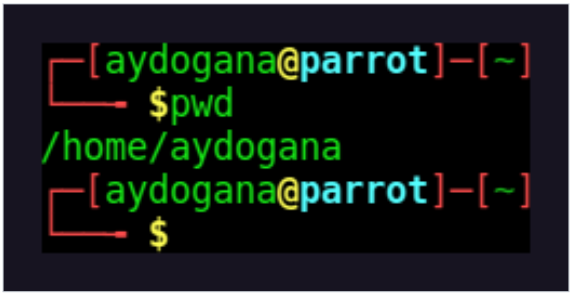
This lab lives in one folder on your Desktop. Before you make it, learn to walk there in the terminal.

Open your terminal (the black `konsole` icon). It opens in your **home** folder. Ask the shell where you are:

Command reference

```
pwd
```

`pwd` means "print working directory," the folder you are standing in. You should see `/home/<you>`.

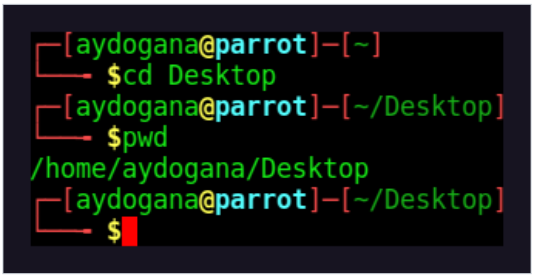


Now step into your Desktop. From your home folder, `Desktop` is right there, so you name it directly. This is a **relative path** (a path from where you already are), and it is easier to type than the full one.

Command reference

```
cd Desktop
pwd
```

`cd` means "change directory." Run `pwd` again to confirm you moved; it now ends in `/Desktop`.



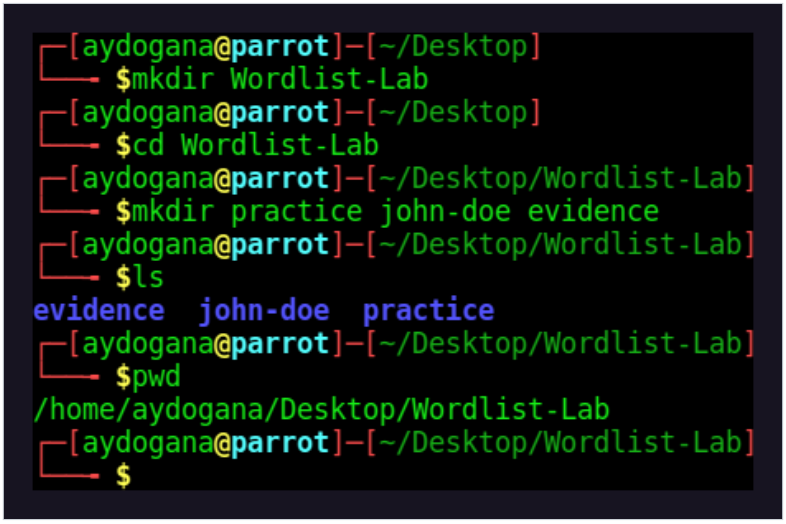
Four things the shell wants you to know. A relative path starts from where you are (`cd Desktop`). **An absolute path** starts from the very top and works anywhere (`cd /home/you/Desktop`). **The tilde** `~` is a shortcut for your home folder, so `~/Desktop` is your Desktop from anywhere. **Two dots** `..` mean "the folder above this one," so `cd ..` steps back up. And Linux is **case sensitive**: `Desktop`, `desktop`, and `DESKTOP` are three different names, so type them exactly.

Now make your workspace and step inside it, then make three subfolders and confirm where you are:

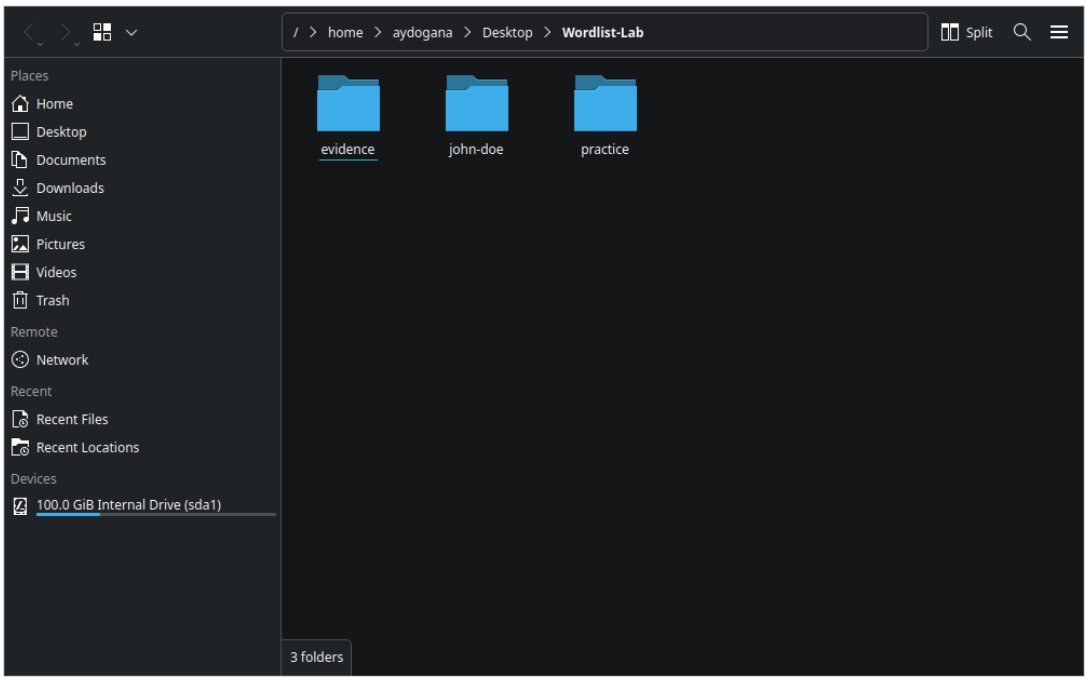
Command reference

```
mkdir Wordlist-Lab
cd Wordlist-Lab
mkdir practice john-doe evidence
ls
pwd
```

`mkdir` makes a directory; `ls` lists what is in the current one. You should see your three new folders, and `pwd` should end in `/Desktop/Wordlist-Lab`.



You can see the same folders in the graphical file manager. Open **Files** (Dolphin) and click through `Home`, `Desktop`, `Wordlist-Lab`. The terminal and the file manager are two windows onto the same disk.



Check your tools and your free space. You will use `crunch` to build lists, `wc`, `head`, `tail`, `ls`, and `du` to inspect them, and `hydra` to test a login. Confirm they are present, and check you have room:

Command reference

```
crunch | head -n 1
hydra -h 2>&1 | head -n 1
john 2>&1 | head -n 1
df -h ~ | tail -n 1
```

```
┌─[aydogana@parrot]─[~/Desktop/Wordlist-Lab]
└──╼ $crunch | head -n 1
crunch version 3.6

Crunch can create a wordlist based on criteria you specify.  The output from crunch can be sent to the screen, file, or to an
other program.

Usage: crunch <min> <max> [options]
where min and max are numbers

Please refer to the man page for instructions and examples on how to use crunch.
┌─[aydogana@parrot]─[~/Desktop/Wordlist-Lab]
└──╼ $hydra -h 2>&1 | head -n 1
Hydra v9.5 (c) 2023 by van Hauser/THC & David Maciejak - Please do not use in military or secret service organizations, or fo
r illegal purposes (this is non-binding, these *** ignore laws and ethics anyway).
┌─[aydogana@parrot]─[~/Desktop/Wordlist-Lab]
└──╼ $john 2>&1 | head -n 1
John the Ripper 1.9.0-jumbo-1+bleeding-aec1328d6c 2021-11-02 10:45:52 +0100 OMP [linux-gnu 64-bit x86_64 AVX2 AC]
┌─[aydogana@parrot]─[~/Desktop/Wordlist-Lab]
└──╼ $df -h ~ | tail -n 1
/dev/sda1       100G   24G   74G  25% /home
┌─[aydogana@parrot]─[~/Desktop/Wordlist-Lab]
└──╼ $
```

Section B. A wordlist of numbers only

Your first list is every possible six digit PIN. Step into the `practice` folder first, so the file lands where you expect:

Command reference

```
cd ~/Desktop/Wordlist-Lab/practice
crunch 6 6 0123456789 -o numbers6.txt
```

Read the arguments: `6 6` is the minimum and maximum length (both six, so every line is exactly

six characters), `0123456789` is the character set, and `-o numbers6.txt` writes the result to that file. Notice what `crunch` tells you before it runs: the exact number of bytes and lines it is about to create.

```
┌[aydogana@parrot]-[~/Desktop/Wordlist-Lab/practice]
└─ $crunch 6 6 0123456789 -o numbers6.txt
Crunch will now generate the following amount of data: 7000000 bytes
6 MB
0 GB
0 TB
0 PB
Crunch will now generate the following number of lines: 1000000

crunch: 100% completed generating output
┌[aydogana@parrot]-[~/Desktop/Wordlist-Lab/practice]
└─ $
```

Look at the first and last lines. Crunch counts in order and **keeps the leading zeros**, so the very first candidate is `000000`, not `0`:

Command reference

```
head -n 3 numbers6.txt
tail -n 3 numbers6.txt
```

```
┌[aydogana@parrot]-[~/Desktop/Wordlist-Lab/practice]
└─ $head -n 3 numbers6.txt
000000
000001
000002
┌[aydogana@parrot]-[~/Desktop/Wordlist-Lab/practice]
└─ $tail -n 3 numbers6.txt
999997
999998
999999
┌[aydogana@parrot]-[~/Desktop/Wordlist-Lab/practice]
└─ $
```

Now measure the file three ways: how many lines, how many bytes, and how big on disk:

Command reference

```
wc -l numbers6.txt
ls -l numbers6.txt
du -h numbers6.txt
```

```
┌[aydogana@parrot]-[~/Desktop/Wordlist-Lab/practice]
└─ $wc -l numbers6.txt
1000000 numbers6.txt
┌[aydogana@parrot]-[~/Desktop/Wordlist-Lab/practice]
└─ $ls -l numbers6.txt
-rw-rw-r-- 1 aydogana aydogana 7000000 Sep 22 02:53 numbers6.txt
┌[aydogana@parrot]-[~/Desktop/Wordlist-Lab/practice]
└─ $du -h numbers6.txt
6.7M    numbers6.txt
┌[aydogana@parrot]-[~/Desktop/Wordlist-Lab/practice]
└─ $
```

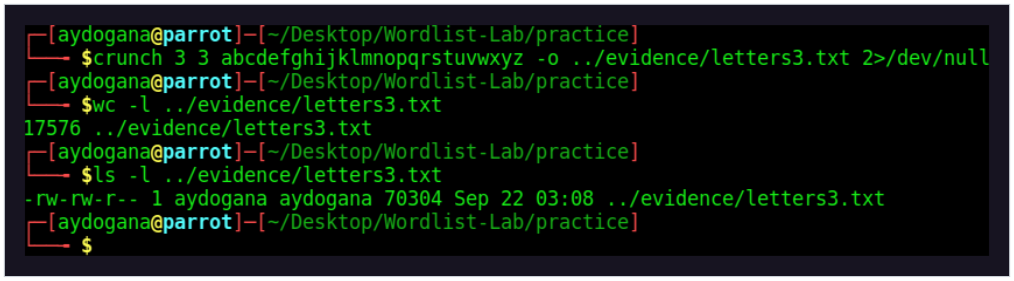
There are exactly 1,000,000 lines, the file is 7,000,000 bytes, and `du` reports about 6.7M. That matches the arithmetic from Part I: 10^6 candidates, each seven bytes long, is seven megabytes, which is 6.68 mebibytes. Record all three numbers in your worksheet.

Section C. Add letters, and learn to count instead of generate

First a small, fast one, so you can see it finish. Every three letter lowercase string goes into your `evidence` folder:

Command reference

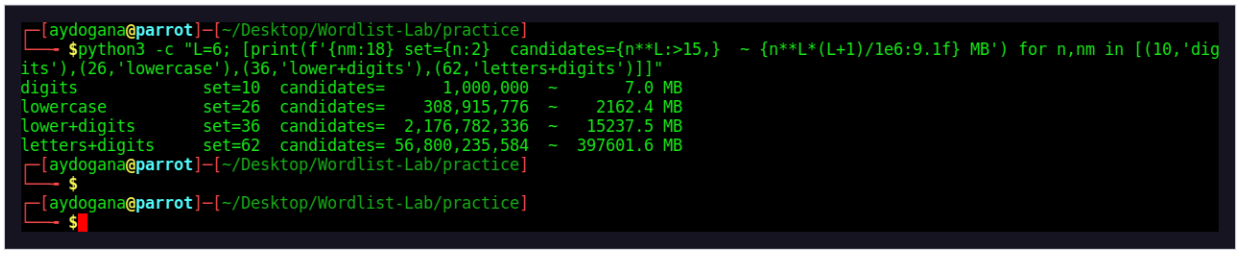
```
crunch 3 3 abcdefghijklmnopqrstuvwxyz -o ../evidence/letters3.txt 2>/dev/null
wc -l ../evidence/letters3.txt
ls -l ../evidence/letters3.txt
```



That is $26^3 = 17{,}576$ lines. Small. But watch what happens when the length grows to six and the character set grows. You will NOT generate these; you will *calculate* them, because some would not fit on your disk. Use Python as a calculator:

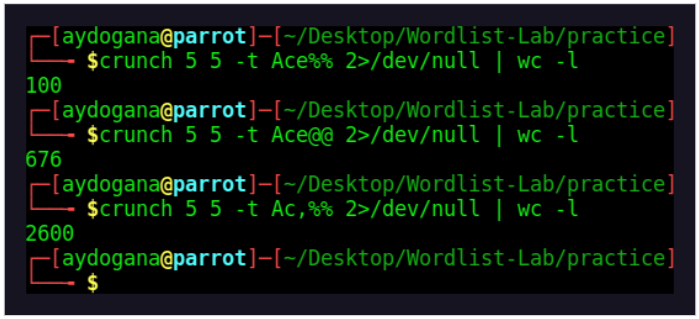
Command reference

```
python3 -c "L=6; [print(f'{nm:18} set={n:2}  candidates={n**L:>15,}  ~ {n**L*(L+1)/1e6:9.1f} MB')
↪  for n,nm in [(10,'digits'),(26,'lowercase'),(36,'lower+digits'),(62,'letters+digits')]]"
```



Six digits was seven megabytes. Six lowercase letters is already about two gigabytes. Six letters and digits together, 62^6, is nearly 57 billion candidates and about 400 gigabytes. The password did not get much longer; the *character set* got wider, and the count exploded.

Fix what you know, vary what you do not. You rarely need every combination. With `crunch`'s `-t` templates you keep part fixed and vary the rest, which shrinks the list enormously. Compare: varying two digits after a fixed word is 100 candidates, two lowercase letters is 676, and one uppercase letter plus two digits is 2,600. The more positions you pin down, the smaller and faster your list.

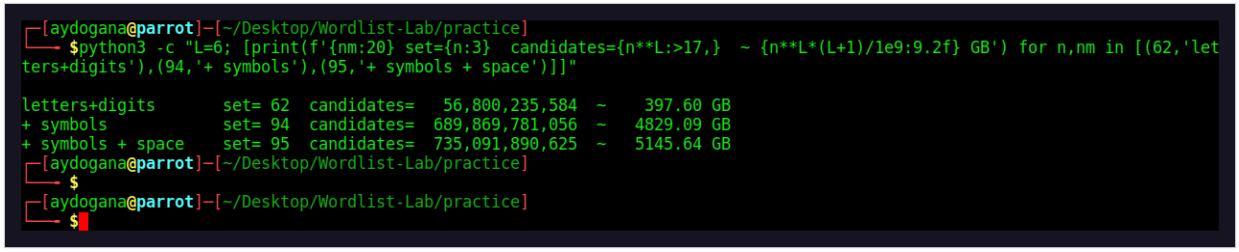


Section D. Add symbols, and see the wall

Symbols make a password stronger by widening the character set again. Add the common keyboard symbols and the count climbs past anything you could store. Calculate it; do not build it:

Command reference

```
python3 -c "L=6; [print(f'{nm:20} set={n:3}  candidates={n**L:>17,}  ~ {n**L*(L+1)/1e9:9.2f} GB')
↪  for n,nm in [(62,'letters+digits'),(94,'+ symbols'),(95,'+ symbols + space')]]"
```



Letters and digits at length six were about 400 GB. Add the roughly 32 printable symbols and you pass 4,800 GB, nearly five terabytes, for a password only six characters long. This is the wall that makes brute force hopeless against a decent password, and it is why the rest of this lab guesses *smart* instead of guessing *everything*.

Do not fill your disk. The numbers above are calculations, not files. Never run `crunch` on a full letters plus symbols set at length six or more without `-o` pointed somewhere with room, and never leave it running. You have already seen `crunch` announce a multi terabyte job; the next section is how you stop one.

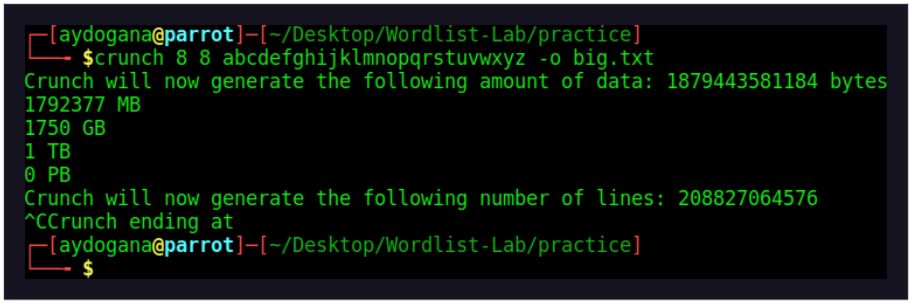
Section E. Stopping a command safely

Type a command that would run for days, so you can practise stopping it. Start generating every eight letter string, watch `crunch` announce a job of nearly two terabytes, then **press `Ctrl+C`** to cancel it:

Command reference

```
crunch 8 8 abcdefghijklmnopqrstuvwxyz -o big.txt
```

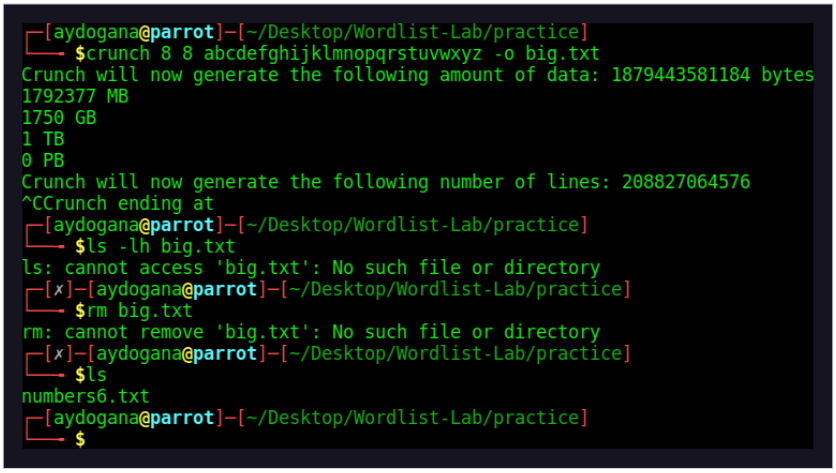
After a second or two, hold `Ctrl` and press `C`. The command stops and your prompt returns.



Now check what it left behind and remove the half written file, so it does not sit on your disk:

Command reference

```
ls -lh big.txt
rm big.txt
ls
```



Three key combinations, three different jobs. `Ctrl+C` sends an interrupt to the running foreground program and stops it; this is the one you use here. `Ctrl+Z` only *suspends* the program (it pauses in the background and is not finished; `fg` would resume it). `Ctrl+X` is NOT a general stop; in most terminals it does nothing, and inside an editor like `nano` it means "exit the editor." When you need to end a running command, reach for `Ctrl+C`.

Section F. A guided example: guessing Mia's password

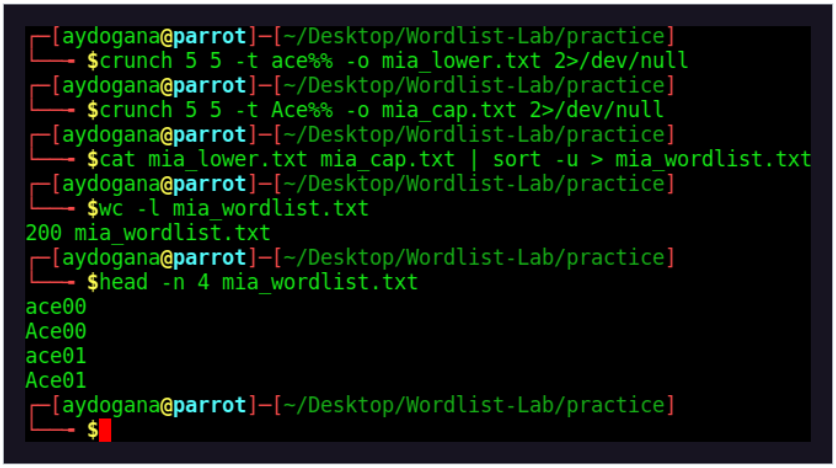
Now the real skill. Meet **Mia Kent**, a fictional person with a public profile. Her page tells you: she was born in **1990**, she has a dog named **Ace**, and she keeps her accounts short. Her account sits on a login panel with the same rules you will meet in Section G: the **username is at most 4 characters** and the **password is exactly 5 characters**, letters and digits only. You will build a small, targeted wordlist from her life and recover her password. Follow every step; you will repeat this method on your own in Section G.

1. Read the life, list the clues. From Mia's page the useful short tokens are her dog `ace` and her birth year `1990` (so the two digit ending `90`). A common human habit is *a name or pet followed by two digits*. Since the password must be exactly five characters, "Ace" (three letters) plus two digits fits perfectly.

2. Build the candidate family with a template. Use `crunch -t` to keep `Ace` fixed and vary only the last two digits, both as she typed it (`Ace`) and all lowercase (`ace`). Then combine the two families and remove duplicates:

Command reference

```
cd ~/Desktop/Wordlist-Lab/practice
crunch 5 5 -t ace%% -o mia_lower.txt 2>/dev/null
crunch 5 5 -t Ace%% -o mia_cap.txt 2>/dev/null
cat mia_lower.txt mia_cap.txt | sort -u > mia_wordlist.txt
wc -l mia_wordlist.txt
head -n 4 mia_wordlist.txt
```



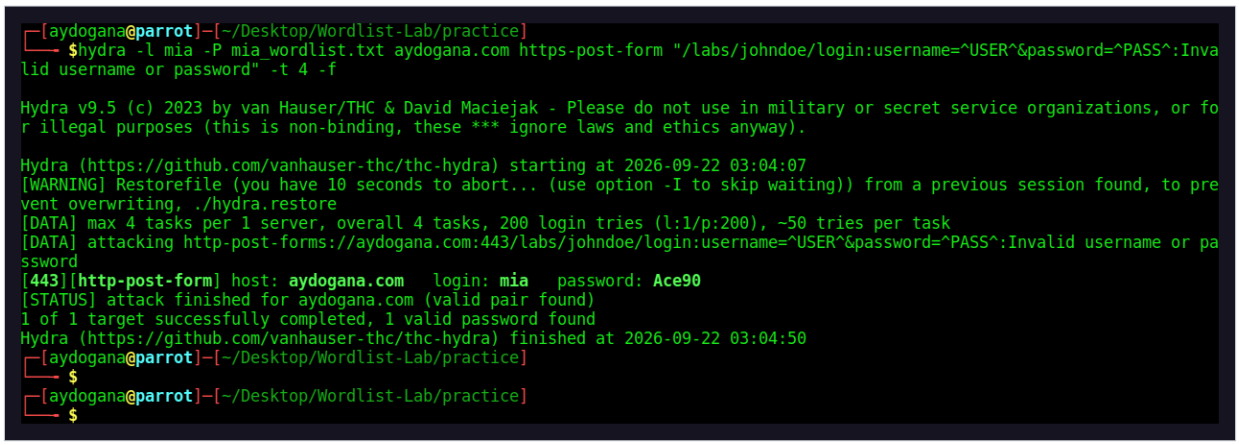
That is only 200 candidates, and every one is exactly five characters using letters and digits, so every one obeys the login's rules. `sort -u` did two jobs at once: it sorted the lines and removed duplicates (the `-u` for "unique"). Building a list that already respects the rules is what makes a targeted attack small.

3. Run the guesses against the panel. The tool `hydra` submits each candidate to the login form and stops when one works. The command names the username (`-l mia`), the wordlist (`-P mia_wordlist.txt`), the site, and the exact form fields, with a failure marker so `hydra` can tell a wrong guess from a right one:

Command reference

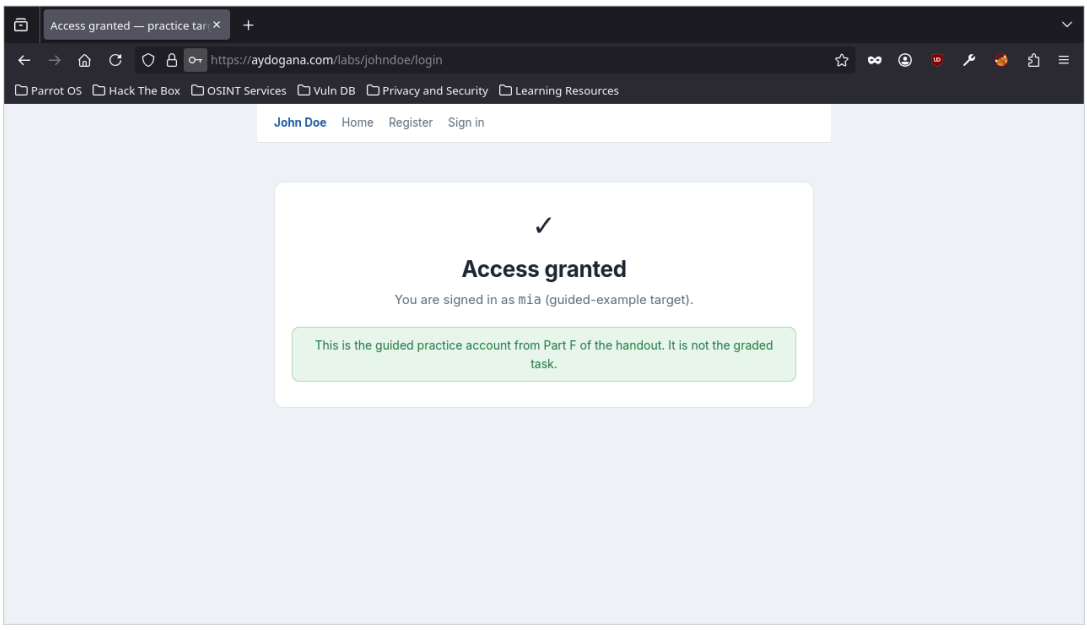
```
hydra -l mia -P mia_wordlist.txt aydogana.com https-post-form
↪  "/labs/johndoe/login:username=^USER^&password=^PASS^:Invalid username or password" -t 4 -f
```

The pieces: `^USER^` and `^PASS^` are where `hydra` inserts each guess; the text after the last colon is the message the page shows on a *failed* login, so any response *without* it is a success; `-t 4` keeps to four attempts at a time (low and polite); and `-f` stops at the first valid pair.



Hydra reports `login: mia password: Ace90`. Her password was her dog plus her birth year, exactly the habit you guessed. A list of two hundred candidates found it in seconds.

4. Confirm it in the browser. Open Firefox, go to the panel, and sign in as `mia` with the password you recovered. You reach the account, which is your proof:

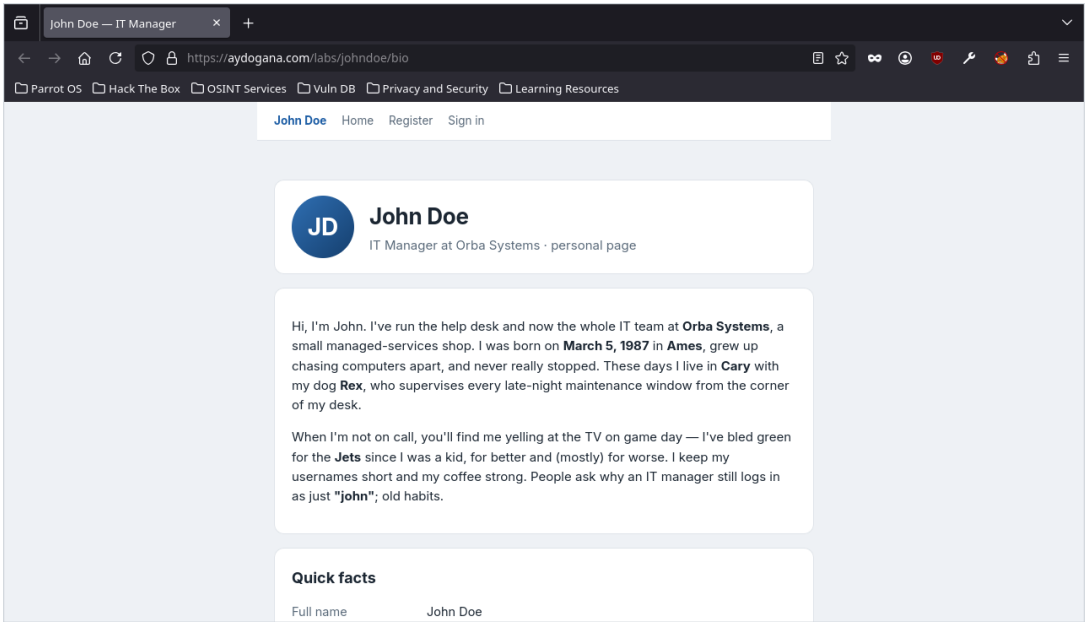


Why each transformation mattered. You added *case variants* (`ace` and `Ace`) because you did not know how she capitalized it. You added *two digits* because a birth year ending is a near universal habit. You *removed duplicates* so you did not waste guesses, and you kept every candidate to *exactly five characters and letters or digits only* so none violated the rules and wasted a try. Each choice made the list either more likely to contain the answer or smaller to run.

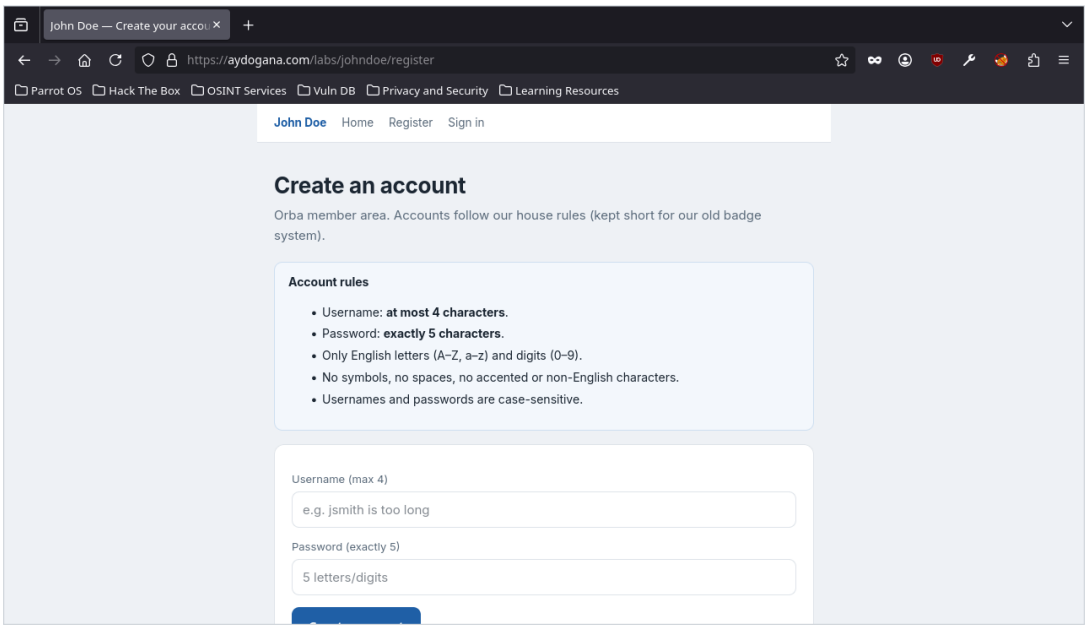
Section G. Your task: recover John Doe's login

Now do it yourself, against a different fictional person. Everything you need is on his site and the sign in panel; the answer is not printed anywhere. Record each part in the worksheet.

1. Read the biography. Open Firefox and visit the John Doe profile page at the address your instructor gives (it is on the same site as the panel). Read it the way you read Mia's: note his name, his dog, his town, his employer, his team, and his date of birth.



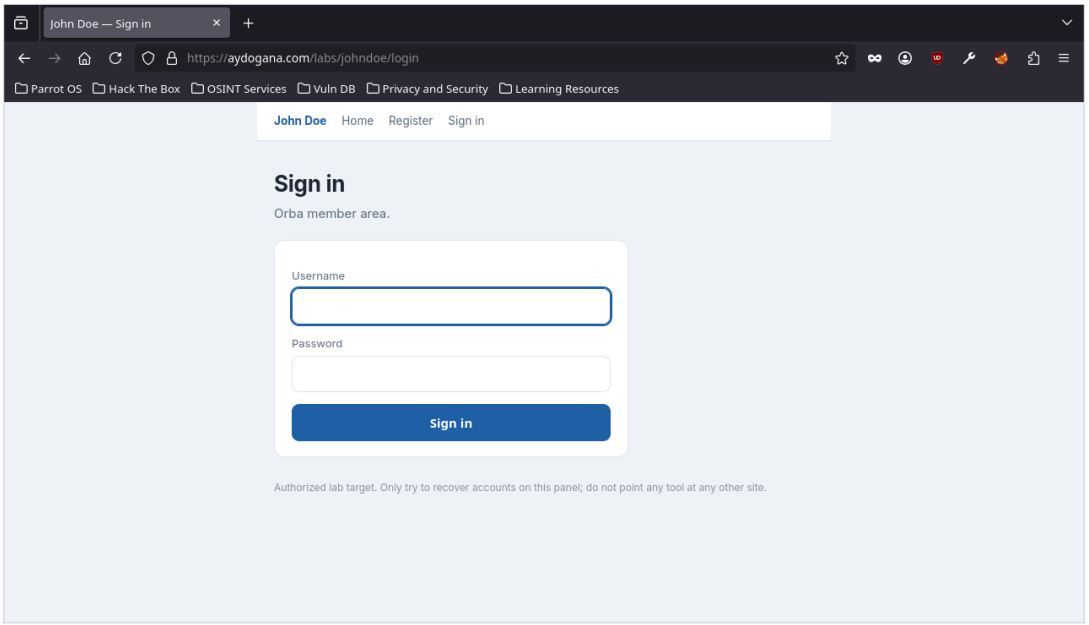
2. Observe the rules. Visit the **Register** page and read the account rules carefully. Write them in your worksheet exactly. They tell you how long a username may be, how long the password must be, and which characters are allowed.



3. Decide your username candidates. The rules cap the username length. From his name, list the short usernames a person like him might use (his first name, his last name, initials). Write your candidates and why.

4. Build a targeted wordlist. Using the biography and the *exact* password rule, pick the tokens most likely to appear (a short name or pet, a year, a place) and build a list with `crunch -t`, exactly as you did for Mia. Keep every candidate to the required length and allowed characters. Combine families and remove duplicates with `sort -u`. Save it in your `john-doe` folder and record the final candidate count.

5. Recover the login. Point `hydra` at the sign in panel with your username and your wordlist, using the same command shape as Section F (change only the username and the wordlist file). Keep `-t` low. When it finds the password, sign in on the panel in Firefox to confirm, and capture the success screen for your worksheet.



Reminder on scope. Your only target is this John Doe training panel. Do not run `hydra` or any other tool against the course portal, the login you use as a student, any real website, or a classmate. If your first wordlist does not contain the password, that is normal: widen your guesses (more case variants, a different two digit ending, a place instead of a pet) and try again. A short, honest write up of what you tried earns

credit even if the recovery takes several attempts.

Section H. Making sense of what you saw

Hold the two halves of this lab side by side. In Section D you calculated that trying *every* six character letters and digits password is about 57 billion candidates and hundreds of gigabytes: hopeless to run. In Section F a *targeted* list of two hundred candidates found a five character password in seconds. The difference was not a better computer; it was knowing the person. Every fact on a public profile, a pet, a birth year, a team, shrinks the search from "every possible password" to "the few this particular human would choose." That is the whole lesson: password strength is not only about length and symbols, it is about being *unpredictable*. A defender answers this attack with long, unrelated passwords, a limit on how many guesses a login will accept, a lockout after repeated failures, and a second factor beyond the password. You will weigh these in the questions below.

3. Analysis questions

Answer these in your own words in the worksheet.

Q1. The count. A six digit numeric wordlist held exactly how many candidates, and how many bytes was the file? Show the arithmetic (n^L for the count, and candidates $\times(L+1)$ for the size), and explain in one sentence why `ls` and `du` reported the size differently.

Q2. Brute force versus dictionary. In your own words, what is the difference between a brute force attack and a dictionary (wordlist) attack? For guessing Mia's password, which did you use, and why was it the right choice?

Q3. The transformations. List the tokens you pulled from John Doe's biography and the transformations you applied (case variants, digit endings, and so on). How many candidates were in your final list, before and after removing duplicates?

Q4. The limits. Your targeted list was tiny compared to every possible password. What one change to his password (longer, unrelated to his life, added symbols) would have kept it out of your list, and what does that change cost him in daily use?

Q5. Defense. You run the login service John Doe uses. Name two defenses that would stop the attack you just ran even if his password stayed weak, and say in one sentence why each one works.

4. Deliverable, grading, and ethics

Deliverable. Fill in the provided Word worksheet, "The Predictable Password - Student Worksheet.docx": record your calculations, your terminal observations, your guided example and John Doe results, and your answers to Q1 to Q5. Save it as `Lastname_Firstname_PredictablePassword.docx` (or export to PDF) and upload it. A screenshot clearly showing your successful John Doe sign in, on the panel in your browser, is the required proof.

Criterion	Points
Section A to E: terminal workspace built; six digit list generated and correctly measured (count, bytes, MB versus MiB)	25
Character set and symbol calculations completed and interpreted (Sections C and D)	15
`Ctrl+C` used to stop a command; the three key combinations explained (Section E)	10
Guided example reproduced: targeted list built and Mia recovered (Section F)	15
John Doe task: targeted wordlist built to the rules, login recovered, success screenshot included (Section G)	20
Q1 to Q5 answered in your own words	15
Total	**100**

Academic integrity, scope, and ethics. Individual work. Run every command from YOUR OWN Parrot machine (a virtual machine is fine). Every biography in this lab is fictional and every target is a training panel your instructor set up for this course; no real person and no real system is involved. Password guessing is legal on a target you own or are explicitly authorized to test; run it against someone else's account without written authorization and it is a crime. Point your tools at the John Doe training panel only, never at the course portal, your student login, a classmate, or any real website. This skill is taught so you can measure password strength, recognize weak choices, and defend against exactly this attack. A correctly reported "I could not recover it yet," with your attempts recorded, earns partial credit; an invented password earns zero. AI may help you understand the tools, but run every command yourself. The technique is neutral; the target and the authorization are what make it right or wrong.

Tools as shipped on Parrot Security, September 2026: crunch 3.6, Hydra 9.5, John the Ripper 1.9.0-jumbo. Questions? Email the instructor at aydogana@uncw.edu.