Problem Solving and Engineering Design part 3

ESAT2A2

Giel Swenters (r1006315)

Crack the Wi-Fi

INDIVIDUAL ASSIGNMENT MID-TERM REPORT

Co-titular
Vincent Rijmen

Coach(es)
John Gaspoz
Dilara Toprakhisar

ACADEMIC YEAR 2024-2025

Declaration of originality

We hereby declare that this submitted draft is entirely our own, subject to feedback and support given us by the didactic team, and subject to lawful cooperation which was agreed with the same didactic team.
Regarding this draft, we also declare that:

1. *Note has been taken of the text on academic integrity (https://eng.kuleuven.be/studeren/masterproef-en-papers/documenten/20161221-academischeintegriteit-okt2016.pdf).*
2. *No plagiarism has been committed as described on https://eng.kuleuven.be/studeren/masterproef-en-papers/plagiaat.*
3. *All experiments, tests, measurements, …, have been performed as described in this draft, and no data or measurement results have been manipulated.*
4. *All sources employed in this draft – including internet sources – have been correctly referenced.*

Contents

List of Figures

List of Tables

Rewritten chapter
with visual changes

2 ~~Sniffer~~Reading the network traffic using a sniffer Ti

To capture and read network traffic on the router, a network capture system, known as a sniffer, is implemented. This system is developed using Scapy, which is "a Python program that enables the user to send, sniff, dissect and forge network packets. This capability allows construction of tools that can probe, scan or attack networks." [Biondi, 2024] In this implementation, the system intercepts and analyzes traffic from the WEP-encrypted network. By combining Scapy with a decryption algorithm~~that uses the WEP key obtained with Aircrack~~, the sniffer captures, decodes and reads network traffic. ~~Notably, the packet capture~~ This decryption algorithm requires the encryption key for the WEP encryption. This key can easily be obtained by the Aircrack attack. Worth noting, the sniffer operates externally to the target network, as illustrated in Figure 2. I1, I3

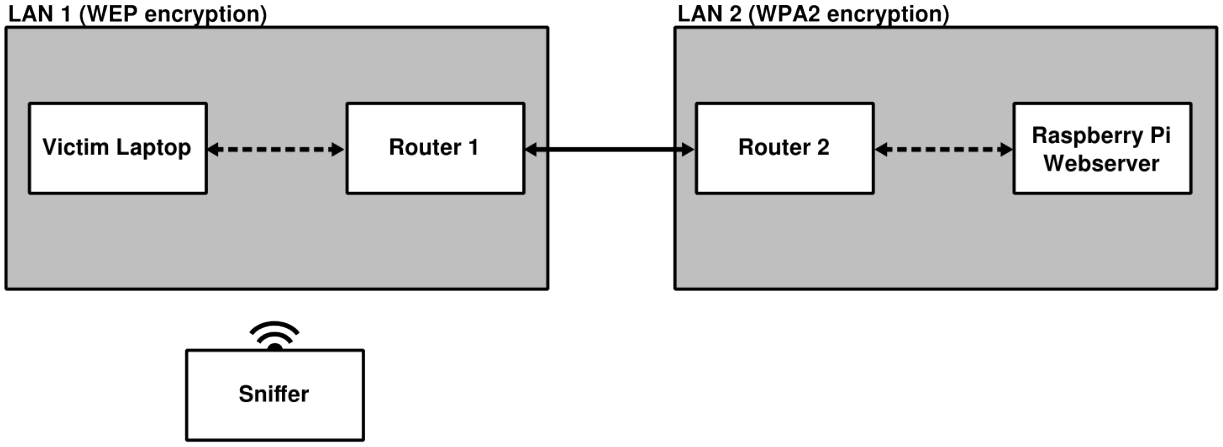

Figure 1: Network Layout

2.1 ~~Sniffing~~

The system utilizes Scapy's "sniff" functionality to intercept network traffic directed at the Raspberry Pi. When using monitor mode, it can detect and sniff all wireless traffic within range. ~~This can be filtered to only include~~ These captured packets can then be filtered based upon the wanted criteria. In this case two filters will be active. First of all, packets gets sent from all different devices. Each device has a unique MAC address, which can be interpreted as a "physical address" for the device. It is important to only filter the packets sent from ~~a specific MAC address.~~ the device, and thus the MAC address, of the victim. A second filter will be applied to the packets type. For this project, only the WEP-encrypted packets are relevant, since those are the packets that can be decrypted. I1, I3

2.1 WEP protocol

Sh1, Me1

This paragraph will give a short history of the WEP protocol and explain how it encrypts its data using the RC4 algorithm. This descryption is kept very simple and the algorithm will be simplified, the purpose of this is to gather insight quicker and more easily. To follow along the explanation, a pseudo-code in python is added to the appendices ??.

(?? is a reference to a different chapter)

WEP stands for Wired Equivalent Privacy, which also describes it's function. This protocol tries to protect your private data in a wireless connection with the same security as a wired connection. In 1999, it was the standard for Wi-Fi security. It makes use of the RC4 encryption algorithm to encrypt the data. On this day, WEP isn't used anymore because flaws were detected and abused. In the configured network used for this report 2, router 1 uses the WEP protocol. This router will be used as a weak point in the network.

The following paragraphs ~~describe~~ describes how the RC4 algorithm works and how to decrypt ~~its packages~~ the packets using the key gathered by the Aircrack attack [chapter ??]. It is divided into 3 parts. The first Sp part describes the key-scheduling algorithm, this is an algorithm which uses the key to create a 256-byte array, also called the S-box. The second part goes over the the algorithm which generates the keystream, using the S-box. In the third part the encryption is explained. I4

2.1.1 RC4 encryption

~~The following~~ This explanation on the RC4 encryption is based upon the article written by S Sriadhi, Robbi Rahim and Ansari Saleh Ahmar [S Sriadhi, 2018]. This explanation is a simplified description, but provides enough insight to understand the use of it in our decryption. **??** gives a pseudocode written based upon the process in this paragraph.

~~The package~~ Key-scheduling algorithm D

The packet before encryption consists of a plain, readable text. To encrypt this ~~package~~packet, a secret key is necessary. This key could be any string of text with a length between 1 and 256 ~~characters~~bytes. In most cases this key is less than 256 ~~characters~~ bytes long. To obtain a string with the desired length of 256, the shorter key is simply repeated. Inh

This 256 ~~character~~ byte long string is now used to generate ~~a keystream~~the S-box. Using the ASCII-table [ASCII-table, 1963], the extended key is converted to a 3 digits long code. This is done for the entire extended key. At the start, the ~~keystream~~ S-box is a simple ~~table~~ array with 256 entries. The first entry holds 0, the one after holds 1. Repeating this results in the last entry holding 255. Using a simple repetitive algorithm, this ~~keystream table~~ array will be permutated using the extended key. Note the following two variables: Inh

- i holds the amount of iterations the algorithm has been done. It starts at 0 and ends at 255.
- j holds ~~an integer~~ a natural number between 0 and 255 which is calculated by formula 4. Using the modulus operator, j will never exceed these boundaries. For the first iteration, j holds a value 0.

These variables are also used as indices for the key or the ~~keystream~~S-box. In this case they are noted key~~(stream)~~[i] or s_box[i].

$$j = (j + \underline{\text{keystream}[j]}\text{s\_box}[j] + \text{key}[i]) \bmod 256 \tag{1}$$

Note that the j on the right side of the equation still holds the value of the previous iteration.

To finish the algorithm, ~~2 entries in the keystream~~ two entries are swapped. These entries are defined by the calculated indices i and j. Tel

$$\underline{\text{keystream}[i]}\text{s\_box}[i], \underline{\text{keystream}[j]}\text{s\_box}[j] = \underline{\text{keystream}[j]}\text{s\_box}[j], \underline{\text{keystream}[i]}\text{s\_box}[i]$$

Note that the swapped values can be any value between 0 and 256 and are only indicated by the indices i and j.

~~After completing the 256 iterations~~ Generating the keystream D

Now that the S-box has been created, the keystream can be generated using another algorithm. This algorithm makes use of three variables i, j and k. Note that these variables will be reset at the start of the algorithm. This means that the value of these variables in the key-scheduling algorithm has no impact on the starting values.

The first two variables serve a similar function as in the key scheduling algorithm: Inh

- i holds the amount of iterations completed plus one. This means it starts with a value of 1 and gets the value of the next natural number each iteration. Note that i is using the modulus 256 operator, hence it will never have a value higher than 255.
- j is once again a natural number calculated by formula 5. It makes use of the values stored in the S-box. ~~The~~ j has the same upper bound due to the same modulus operator.

$$j = (j + \text{s\_box}[i]) \bmod 256 \tag{2}$$

On the right side of the equation, j holds the value of the previous iteration. This was also the case in the previous algorithm.

With these two variables defined, the next thing to look at is the desired lenght of the keystream. RC4 uses a one-on-one encryption method, which will be discussed in the third part. This means that the desired lenght is the same lenght as the message that has to be encrypted. To ensure this, the algorithm will be executed one time for each character in the message. Inh

Every iteration, a new value for the third variable, k, will be calculated as follows:

$$k = (s\_box[i]+s\_box[j]) \bmod 256 \tag{3}$$

Once again the modulus operator has been used as this variable will be used as an index. More specifically, the byte at the entry k in the S-box, or $\text{s\_box}[k]$, will be added at the end of the keystream. Note that the keystream empty at the beginning. The iteration ends with the same swapping operation that has been discussed earlier. Inh

$$\text{s\_box}[i], \text{s\_box}[j] = \text{s\_box}[j], \text{s\_box}[i]$$

After completing this process, the final keystream is generated. Using this keystream, the original plain text can now be encrypted.

Encryption of the packet D

This process starts with converting the ASCII codes ~~of each character to an~~ to 8 digit long binary ~~code~~codes, or bytes. This is done for the keystream and for the plain text. Next, the logical operator XOR is used on both ~~binary strings~~ bytes to generate the encrypted output string, which also is ~~an 8 digits long string for each character~~one byte.

The visualization in table 2 gives an example of the process described in the paragraph above. The keystream used in this example is completely random.

INPUT			KEYSTREAM			OUTPUT
index	*~~character~~byte*	*binary*	*index*	*~~character~~byte*	*binary*	*binary*
0	P	01010000	0	¥	10100101	11110101
1		10100000	1	D	01000100	11100100
2	e	01100101	2	c	00101110	01001011
3	n	01101110	3	L	01001100	00100010
4		10100000	4	š	10011010	00111010
5	O	01001111	5	ö	11110110	10111001

F1, F3, G

Table 1: Example of XOR-encryption using a random keystream

One of the benefits of using the XOR operator is how easy it is to reverse. To reverse the encryption, use the XOR operator on the encrypted input (or the output in 2) and the keystream. This is why XOR is called ”a symmetrical operator”.

The final step of the encryption is to turn the final binary output back into symbols using the ASCII-table [ASCII-table, 1963].

2.1.2 RC4 decryption

As mentioned in the introduction 2.2, the way the intercepted ~~packages~~ packets are decrypted will be the exact same way as the victim normally would decrypt them. This is possible because the secret key is known as a result of the successful Aircrack attack **??**.

First, the key is used to generate the same keystream as described in the encryption process. Then, the encrypted message and the keystream are both converted to binary. When converted, the XOR operator is used on both binary streams to generate the decrypted binary stream. This illustrates how easily reversible the XOR operator is, as explained above. The last step is to convert the binary stream back to the decrypted message.

Rewritten chapter
without tracking the changes

Reading the network traffic using a sniffer

To capture and read network traffic on the router, a network capture system, known as a sniffer, is implemented. This system is developed using Scapy, which is "a Python program that enables the user to send, sniff, dissect and forge network packets. This capability allows construction of tools that can probe, scan or attack networks." [Biondi, 2024] In this implementation, the system intercepts and analyzes traffic from the WEP-encrypted network. By combining Scapy with a decryption algorithm, the sniffer captures, decodes and reads network traffic. This decryption algorithm requires the encryption key for the WEP encryption. This key can easily be obtained by the Aircrack attack. Worth noting, the sniffer operates externally to the target network, as illustrated in Figure 2.

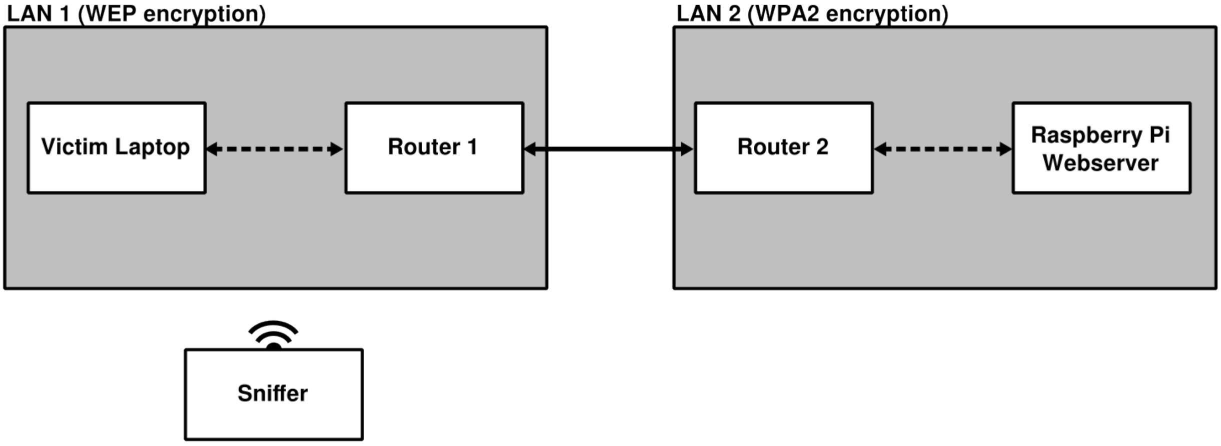

Figure 2: Network Layout

The system utilizes Scapy's "sniff" functionality to intercept network traffic directed at the Raspberry Pi. When using monitor mode, it can detect and sniff all wireless traffic within range. These captured packets can then be filtered based upon the wanted criteria. In this case two filters will be active. First of all, packets gets sent from all different devices. Each device has a unique MAC address, which can be interpreted as a "physical address" for the device. It is important to only filter the packets sent from the device, and thus the MAC address, of the victim. A second filter will be applied to the packets type. For this project, only the WEP-encrypted packets are relevant, since those are the packets that can be decrypted.

2.2 WEP protocol

This paragraph will give a short history of the WEP protocol and explain how it encrypts its data using the RC4 algorithm. This descryption is kept very simple and the algorithm will be simplified, the purpose of this is to gather insight quicker and more easily. To follow along the explanation, a pseudo-code in python is added to the appendices **??**.

WEP stands for Wired Equivalent Privacy, which also describes it's function. This protocol tries to protect your private data in a wireless connection with the same security as a wired connection. In 1999, it was the standard for Wi-Fi security. It makes use of the RC4 encryption algorithm to encrypt the data. On this day, WEP isn't used anymore because flaws were detected and abused. In the configured network used for this report 2, router 1 uses the WEP protocol. This router will be used as a weak point in the network.

The following paragraphs describes how the RC4 algorithm works and how to decrypt the packets using the key gathered by the Aircrack attack [chapter **??**]. It is divided into 3 parts. The first part describes the key-scheduling algorithm, this is an algorithm which uses the key to create a 256-byte array, also called the S-box. The second part goes over the the algorithm which generates the keystream, using the S-box. In the third part the encryption is explained.

2.2.1 RC4 encryption

This explanation on the RC4 encryption is based upon the article written by S Sriadhi, Robbi Rahim and Ansari Saleh Ahmar [S Sriadhi, 2018]. This explanation is a simplified description, but provides enough insight to understand the use of it in our decryption. **??** gives a pseudocode written based upon the process in this paragraph.

Key-scheduling algorithm

The packet before encryption consists of a plain, readable text. To encrypt this packet, a secret key is necessary. This key could be any string of text with a length between 1 and 256 bytes. In most cases this key is less than 256 bytes long. To obtain a string with the desired length of 256, the shorter key is simply repeated.

This 256 byte long string is now used to generate the S-box. Using the ASCII-table [ASCII-table, 1963], the extended key is converted to a 3 digits long code. This is done for the entire extended key. At the start, the S-box is a simple array with 256 entries. The first entry holds 0, the one after holds 1. Repeating this results in the last entry holding 255. Using a simple repetitive algorithm, this array will be permutated using the extended key. Note the following two variables:

- i holds the amount of iterations the algorithm has been done. It starts at 0 and ends at 255.
- j holds a natural number between 0 and 255 which is calculated by formula 4. Using the modulus operator, j will never exceed these boundaries. For the first iteration, j holds a value 0.

These variables are also used as indices for the key or the S-box. In this case they are noted key[i] or s_box[i].

$$j = (j + \text{s\_box}[j] + \text{key}[i]) \bmod 256 \tag{4}$$

Note that the j on the right side of the equation still holds the value of the previous iteration.

To finish the algorithm, two entries are swapped. These entries are defined by the calculated indices i and j.

$$\text{s\_box}[i], \text{s\_box}[j] = \text{s\_box}[j], \text{s\_box}[i]$$

Note that the swapped values can be any value between 0 and 256 and are only indicated by the indices i and j.

Generating the keystream

Now that the S-box has been created, the keystream can be generated using another algorithm. This algorithm makes use of three variables i, j and k. Note that these variables will be reset at the start of the algorithm. This means that the value of these variables in the key-scheduling algorithm has no impact on the starting values.

The first two variables serve a similar function as in the key scheduling algorithm:

- i holds the amount of iterations completed plus one. This means it starts with a value of 1 and gets the value of the next natural number each iteration. Note that i is using the modulus 256 operator, hence it will never have a value higher than 255.
- j is once again a natural number calculated by formula 5. It makes use of the values stored in the S-box. j has the same upper bound due to the same modulus operator.

$$j = (j + \text{s\_box}[i]) \bmod 256 \tag{5}$$

On the right side of the equation, j holds the value of the previous iteration. This was also the case in the previous algorithm.

With these two variables defined, the next thing to look at is the desired lenght of the keystream. RC4 uses a one-on-one encryption method, which will be discussed in the third part. This means that the desired lenght is the same lenght as the message that has to be encrypted. To ensure this, the algorithm will be executed one time for each character in the message.

Every iteration, a new value for the third variable, k, will be calculated as follows:

$$k = (s\_box[i] + s\_box[j]) \bmod 256 \tag{6}$$

Once again the modulus operator has been used as this variable will be used as an index. More specifically, the byte at the entry k in the S-box, or s_box[k], will be added at the end of the keystream. Note that the keystream empty at the beginning. The iteration ends with the same swapping operation that has been discussed earlier.

$$\mathrm{s\_box}[i], \mathrm{s\_box}[j] = \mathrm{s\_box}[j], \mathrm{s\_box}[i]$$

After completing this process, the final keystream is generated. Using this keystream, the original plain text can now be encrypted.

Encryption of the packet

This process starts with converting the ASCII codes to 8 digit long binary codes, or bytes. This is done for the keystream and for the plain text. Next, the logical operator XOR is used on both bytes to generate the encrypted output string, which also is one byte.

The visualization in table 2 gives an example of the process described in the paragraph above. The keystream used in this example is completely random.

INPUT			**KEYSTREAM**			**OUTPUT**
index	*byte*	*binary*	*index*	*byte*	*binary*	*binary*
0	P	01010000	0	¥	10100101	11110101
1		10100000	1	D	01000100	11100100
2	e	01100101	2	c	00101110	01001011
3	n	01101110	3	L	01001100	00100010
4		10100000	4	š	10011010	00111010
5	O	01001111	5	ö	11110110	10111001

Table 2: Example of XOR-encryption using a random keystream

One of the benefits of using the XOR operator is how easy it is to reverse. To reverse the encryption, use the XOR operator on the encrypted input (or the output in 2) and the keystream. This is why XOR is called ”a symmetrical operator”.

The final step of the encryption is to turn the final binary output back into symbols using the ASCII-table [ASCII-table, 1963].

2.2.2 RC4 decryption

As mentioned in the introduction 2.2, the way the intercepted packets are decrypted will be the exact same way as the victim normally would decrypt them. This is possible because the secret key is known as a result of the successful Aircrack attack **??**.

First, the key is used to generate the same keystream as described in the encryption process. Then, the encrypted message and the keystream are both converted to binary. When converted, the XOR operator is used on both binary streams to generate the decrypted binary stream. This illustrates how easily reversible the XOR operator is, as explained above. The last step is to convert the binary stream back to the decrypted message.

References

[ASCII-table, 1963] ASCII-table (1963). `https://www.ascii-code.com/`.

[Biondi, 2024] Biondi, P. (2024). Introduction — scapy 2.6.0 documentation. `https://scapy.readthedocs.io/en/latest/introduction.html`.

[S Sriadhi, 2018] S Sriadhi, Robbi Rahim, A. S. A. (2018). Conf. ser. 1028 012057. *Journal of Physics*.