Graz University of Technology
Institute for Applied Information Processing and Communications - IAIK
Inffeldgasse 16a
A–8010 GRAZ

YAWCU Documentation

Softwareentwicklung und Wissensmanagment
Bakk. Seminararbeit

Johannes Anderwald, 0130975
Reinhard Fellner, 0230269
Tutor: DI Thomas Popp
June 30, 2005

Contents

1 Glossary

Glossary

AP ..... Access Point
BSOD ..... BlueScreen Of Death
BSSID ..... Basic Service Set Identifier
DLL ..... Dynamic Link Library
GUID ..... Globally Unique Identifier
IAIK ..... Institut für Angewandte Informationsverarbeitung und Kommunikationstechnologie
NIC ..... Network Interface Card
YAWCU ..... Yet Another Wired Equivalent Privacy Cracking Utility

2 Introduction

YAWCU is a WEP cracking utility. It is developed by Johannes Anderwald and Reinhard Fellner to prove the insecurity of the WEP encryption algorithm.

- Reading / Writing Pcap capture files
- Decrypting Pcap capture files
- Capturing packets live from an NIC[1]
- Decrypting packets on the fly.
- Key guessing using the basic / improved / KoreK / radius / fudge method

2.1 Software Requirements

YAWCU requires the following packages to run:

- Operating System: Windows NT / 2000 / XP
- Winpcap Library available from http://winpcap.polito.it
- Wildpackets AeroPeek NX available from http://www.wildpackets.com

2.2 Hardware Requirements

YAWCU requires about 2-10 MB RAM to run. YAWCU has been optimized for the Intel Pentium 2+ (i686) platform. A fast CPU is recommended when YAWCU decrypts an capture dump. An AMD-2600 system with 512 MB RAM can decrypt about ~ 160.000 packets per sec.

2.3 Live Capture Requirements

Unfortunately, there exist no open source Windows driver for WLAN NIC. In addition, Windows drivers cut off the IEEE 802.11 Header which results that only the Ethernet frame is available for capturing. However, you need the IEEE 802.11 header in order to decrypt packets because it contains the WEP params section. Furthermore, WLAN cards need to be set into a special mode called **demo** mode. In this mode, WLAN cards receive all IEEE 802.11 frames, even those not indented for the WLAN card. This mode is required to capture the full traffic and eventually decipher the WEP key.

[1] This is achieved via the Wildpackets AeroPeek TM capture framework

Luckily, there is a solution to the above described problems. Wildpackets has written a driver which supports capturing in the demo mode. Furthermore, it has provided a capture framework which facilities the capture process. YAWCU uses this framework to support live capturing for a WLAN NIC. In order to make use of live capturing, YAWCU requires *peek.dll* and *peek5.sys* from Wildpacket's AeroPeek NX installation directory. In addition, you need WLAN card which is supported by the Wildpacket's Aeropeek NX. You can lookup a list of supported adapters at http://www.wildpackets.com/support/drivers

YAWCU has been tested with a Netgear WAG511 802.a/b/g card. However, the Wildpacket driver is very poorly written, which results in repeated BSOD. If you want to use YAWCU live capturing ability, please follow the next guidelines for safe operation:

- If you use Atheros or similar card, use the driver version 3.0.0.111 available from the Wildpacket's website. The older version contains a bug which results in useless capture dumps.
- Don't attempt to remove the card while the operating system is running. It will cause a BSOD. The system will also crash even when Safe-Removal of Devices is used.
- If YAWCU does not respond immediately to an Ctrl-C, wait a few seconds before retrying. If that does not help, close all your open files ;-)
- Don't start more than one capture session at once.
- If YAWCU doesn't capture any packets within the first 10 seconds, then it will very likely never capture any packets within this session. Restart YAWCU to resolve this situation.

2.4 YAWCU Commandline Options

YAWCU supports the following command line options:

- **–conf=config_file** ... specifies the configuration file
- **–list-sources** ... lists the internal names of available PacketSources
- **–list-filters** ... lists the internal names of available PacketFilters
- **–list-interfaces** ... lists available capture interfaces identified by the Winpcap Library.

3 Software Architecture

YAWCU consists primarily of one or more packet processing queues running parallel. The objective of the processing queue is to read packets from a packet source and deliver the packets the packet filters.

3.1 PacketSource Module

The PacketSource is an interface for packet sources. It is used to abstract various types of PacketSources. For more information, see chapter **Extending YAWCU**.

3.2 PacketFilter Module

The PacketFilter is an interface for packet filters. A packet filter receives packets and attempts to process them. If it cannot process it, it can discard the packet. However, the packet filter can also store them and reprocess them at a other time.

3.3 PacketProcessingQueue Module

The PacketProcessingQueue is the bridge between the PacketSource and the PacketFilters. The PacketProcessingQueue is responsible for reading packets and delivering them to the PacketFilters. Figure 1 shows the basic processing loop of the PacketProcessingQueue:

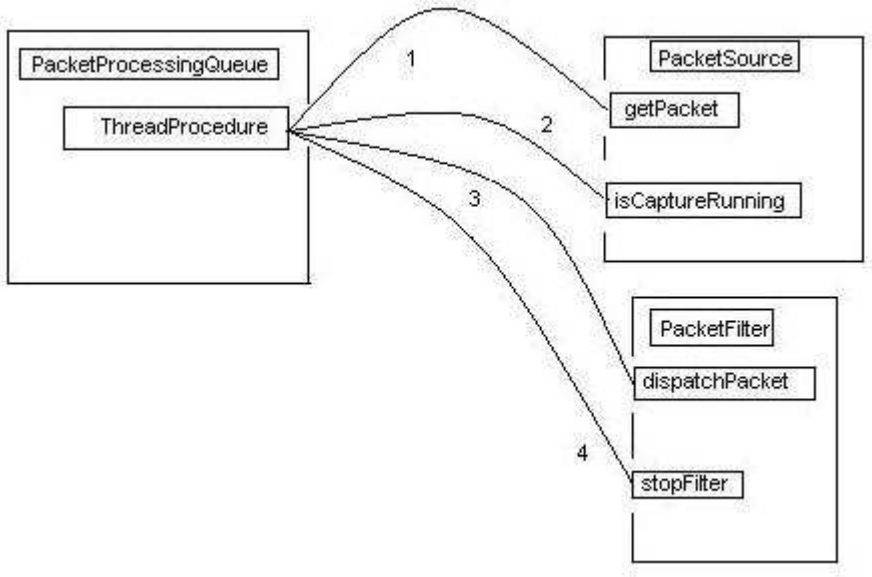

Figure 1:

1. Step 1: The PacketProcessingQueue attempts to read a packet from the PacketSource. If it succeeds, it goes to step 3 otherwise it jumps to step 2

2. Step 2: The PacketProcessingQueue checks if the PacketSource is still running. If it is, then it goes back to step 1. Otherwise it goes to step 4.

3. Step 3: The PacketProcessingQueue delivers the packet to each PacketFilter in its internal queue. If there is no filter, then ThreadProcedure goes to step 4.

4. Step 4: The PacketProcessingQueue stops all filter in its internal queue and the notifies the PacketProcessingManager that it has stopped.

3.4 Multithreading Architecture

YAWCU has the ability of capturing for multiple sources at once. In order to achieve best performance, YAWCU uses a thread for each source. The thread is in embedded in the class PacketProcessingQueue.

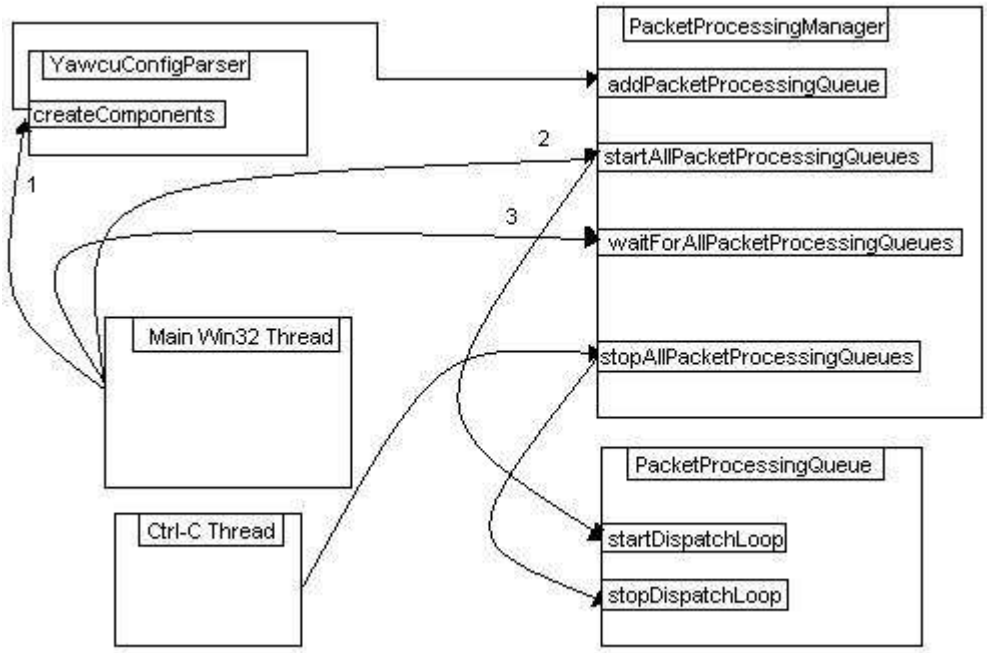

Figure 2:

1. On startup, YAWCU calls the YawcuConfigParser which creates the requested PacketSource and PacketFilter objects. In the next step, these objects are added to the PacketProcessingQueues. The chapter **Configuration Format** provides more information on how this is done.

2. Then the MainThread informs the PacketProcessingManager to start all PacketProcessingQueues

3. The MainThread calls waitForAllPacketProcessingQueues which makes the *MainThread* sleep.

4. If the user presses Ctrl-C, then the *MainThread* is reawakened. The Main Thread then stops all PacketProcessingQueues and terminates causing the program to exit.

Note: If a PacketProcessingQueue stops, it notifies the MainThread that it has stopped. In that case, the MainThread removes the stopped PacketProcessingQueue. If there are still PacketProcessingQueues active, then the MainThread goes to sleep again...

4 Configuration Format

As already stated, YAWCU requires a config file which is used to configure YAWCU to listen on the requested interfaces or to read from a capture dump.
The file configuration format is written in a XML-like syntax. In general, the configuration file consists of *source descriptors* and *filter descriptors*. The following subchaptes describe those two descriptors:

4.1 Source Descriptor

Each source descriptor starts with the tag **<source>** and ends with the tag **</source>**. The following descriptor tags can be placed between:

- sourcename: The sourcename is the internal name of a source. This value is choosen by YAWCU. In order to get a list of available sources, see section *Command Line Options*

- sourcesymbol: The sourcesymbol is tag whose value can be set to anything but cannot be empty. The value of this tag must be unique in comparison to other sourcesymbol values because that value identifies a source in the YAWCU. In addition, filter descriptors reference this value in their *filtersource* tag.

- sourceconf: The sourceconf tag contains configuration arguments for the source. The syntax of the configuration arguments can be looked up in the chapter **Source Interfaces**

- sourcelimit: The sourcelimit tag specifies a maximum of packets which the source interface will attempt to capture / read and then will stop. If this tag is missing, then the source will capture infinitely or read the file untill end of the file.

4.1.1 Source Descriptor Example 1

The below example reads the first 100.000 packets from the dump file **test.pcap**. If there are less packets available, it stops before. If you want to read the complete file, then remove the **sourcelimit** tag.

```
<source>
<sourcename>WpcapSource</sourcename>
<sourcesymbol>wpcap</sourcesymbol>
<sourceconf>caps\test.pcap</sourceconf>
<sourcelimit>100000</sourcelimit>
</source>
```

4.1.2 Source Descriptor Example 2

The below example captures 10.000.000 packets from the device \\Device \\{D397E13F-7BC6-402C-A317-818F8881652B}. If you want to capture infinitely[2], remove the tag **sourcelimit**.

```
<source>
<sourcename>PeekSource</sourcename>
<sourcesymbol>peek</sourcesymbol>
<sourceconf>\\Device\\{D397E13F-7BC6-402C-A317-818F8881652B}</sourceconf>
<sourcelimit>10000000</sourcelimit>
</source>
```

4.2 Filter Descriptor

Each filter descriptor starts with the tag **<filter >** and ends with the tag **</filter>**. The following descriptor tags can be placed between:

- filtername: The filtername is the internal name of a filter. This value is chosen by YAWCU. See chapter **Packet Filter** for list of available filters.
- filtersymbol: The filtersymbol tag uniquely identifies a filter. No other filter descriptor can have the value of this tag.
- filterconf: The filterconf tag passes configuration options to the filter. See chapter **Packet Filter for available configuration options**
- filtersource: The filtersource tag defines on which source the filter will be acting on. The value must match the value of an existing source descriptor's sourcesymbol value.

[2]YAWCU will capture untill Ctrl-C is pressed

- filterparent: The filterparent tag attaches the filter to an existing filter. The value must match the value of an existing filter descriptor's filtersymbol value. If the this tag is missing, then filter will be attached directly to the packet source.

4.2.1 Filter Descriptor Example

This filter example creates two filters: **WeakIVFilter** and **PcapWriteFilter**. Both filters are working in the PacketProcessingQueue **peek**[3]. The WeakIVFilter is attached directly to the PacketProcessingQueue where as the PcapWriteFilter is attached to the WeakIV-Filter and receives only packets from the WeakIVFilter.
Note: In order to be a valid YAWCU config file, you need a source Descriptor which must be placed in front of a filter descriptor.

```
<filter>
<filtername>WeakIVFilter</filtername>
<filtersymbol>weakiv</filtersymbol>
<filterconf>13</filterconf>
<filtermodel>single</filtermodel>
<filtersource>peek</filtersource>
</filter>
<filter>
<filtername>PcapWriteFilter</filtername>
<filtersymbol>writefilter</filtersymbol>
<filterconf>output.pcap|REPLACE|</filterconf>
<filtermodel>single</filtermodel>
<filtersource>peek</filtersource>
<filterparent>dupfilter</filterparent>
</filter>
```

5 Packet Source Interfaces

YAWCU currently supports two interfaces, which are available for capturing:

5.1 WpcapSource Interface

The WpcapSource interface uses the Winpcap Libraryfor reading capture dumps in the Pcap file format. However it also supports capturing data from NICs.

[3]See Source Descriptor Example 2

5.1.1 WpcapSource Configuration

The WpcapSource needs a path to the capture dump file. Therefore the *sourceconf* looks similar to this: *<sourceconf >d:\mydump.pcap</sourceconf>*. If you want to provide multiple paths, then you have to seperate each path with an **";"** without the double quotes. Note: if you want to capture from a NIC, you can either specify the path to the device which looks similar to \\Device \\{ GUID}. The GUID is a number uniquely identifying a device. Alternatively you specify the NIC index. The NIC index depends on the hardware configuration. See chapter *Commandline Options* for more information on how to obtain the correct NIC index.

5.2 PeekSource Interface

The PeekSource interface uses the driver from Wildpacket's AiroPeek NX to capture data. It uses the **peek.dll** and **peek5.sys** from AiroPeek NX installation directory to capture packets.

5.2.1 PeekSource Configuration

The PeekSource interface needs a path to NIC device which is in the form of: \\Device \\{ GUID }.

Note: YAWCU will per default listen on the adapter channel 7 for packets. If you want YAWCU to listen on a different channel, add "channel=X" add the end of the configuration argument. The X can be a channel number between 1-13.

Note: The device path must point to an device which is driven by an Wildpacket driver. If the path is invalid or not directed by a Wildpacket driver, then PeekSource interface will fail to start.

6 PacketFilter Interfaces

This chapter provides information about existing PacketFilters

6.1 WeakIVFilter

The WeakIVFilter only passes on Packets which are of type IEEE 802.11 Data Packets. The packets also have to be encrypted and must have a weak initialization vector. That is first init vector >= 3 and <keylength + 3. In addition, the second init vector must be 255.
The WeakIVFilter accepts as the filterconf parameter either number 5 which is 40/64 WEP key or 13 which is 128 / 158 WEP key, i.e: *<filterconf >13</filterconf>*

6.2 PcapWriteFilter

The PcapWriteFilter writes all receives packets in capture file and redispatches the packets to the next filter in the queue.
The PcapWriteFilter needs 2 parameters as the filterconf param: path to file and howto open the file. It supports three opening methods:

- The |*OPEN*| method only opens the file for writing if it does not exist yet. If does already exist, the PcapWriteFilter will fail to start.
- The |*REPLACE*| method will open any file and replace any file if it exists.
- The |*APPEND*| method will append all packets at the end of the file.

6.3 PacketPerformanceFilter

The PacketPerformanceFilter only measures how many packets have been processed in a given interval. After the interval has elapsed, the PacketPerformanceFilter prints out how many packets have received in a given interval. When the PacketPerformanceFilter is stopped, it prints out information on how many packets have been processed, average packets per sec, and the maximum, - minimum per sec.
The filterconf parameter requires a number which is interpreted in milliseconds. Therefore an interval of 1000 will print out statistics after each second.

For example: *<filterconf >interval=2000</filterconf>* this will print out statistics every 2 seconds.

Note: If the PacketPerformanceFilter is used during live capture, it is highly recommended to set the interval to ˜10000 because the too much information slows down YAWCU enormously.

6.4 IEEE802DataFilter

The IEEE802DataFilter only passes on packets which are of type IEEE 802.11 data packets. It does not any arguments. However the filterconf tag is not allowed to be missing.

6.5 DecryptWEPFilter

The DecryptWEPFilter decrypts encrypted IEEE 802.11 data packets using a key. The DecryptWEPFilter can decrypt packets by using a key from the config file or when it receives a special packet containing the WEP key.
The WEP key must be written in the hexadecimal format. For example: *<filterconf >3132333435</filterconf>*. This parameter is interpreted as 0x31, 0x32, 0x33, 0x34, 0x35.

6.6 BSSIDFilter

The BSSIDFilter passes on packets which match a given criteria:

- The option **eq=BSSID** only passes on packets on which have a given BSSID.
- The option **neq=BSSID** only passes on packets which don't have that BSSID.

For example: *<filterconf>neq=00095D53DF9A</filterconf>* only passes on packets which don't belong to the BSSID 00:09:5D:53:DF:9A

6.7 MACFilter

The MACFilter only passes on packets where the MAC address matches the following criteria:

- The option **sa=MAC** only passes on packets where the sender has that given MAC.
- The option **nsa=MAC** only passes on packets where the sender does not have that given MAC.
- The option **ra=MAC** only passes on packets where the recipient has that given MAC.
- The option **nra=MAC** only passes on packets where the recipient does not have that given MAC.
- The option **sra=MAC** only passes on packets where the recipient or the sender has that given MAC.
- The option **nsra=MAC** only passes on packets where the recipient and the sender does not have that given MAC.

For example: *<filterconf>sa=00095D53DF9A</filterconf>* only passes on packets where the sender has the MAC address 00:09:5D:53:DF:9A

6.8 ListBSSIDFilter

The ListBSSIDFilter prints out statistics about BSSID. Everytime a new BSSID is found prints an informal message and redispatches the packet to attached subfilters.
If the filerconf contains a valid file path, it writes that information to a log file.
For example: *<filterconf>bssid-log.txt</filterconf>* writes all discovered BSSID to the log file bssid-log.txt

7 Extending YAWCU

This chapter is intended for those who are interested on working on YAWCU and extending its features.

7.1 PacketFilter

In order to add a new PacketFilter, a new PacketFilter must implement the following interface:

7.1.1 PacketFilter Interface

7.1.2 Additional Information

This chapter contains additional information for PacketFilter writers:

1. Reduce the amount of debug information. Too much debug information can slow down YAWCU dramatically.
2. If debug information is required, write the information to a file. Override the processFilterEvent function to correctly close the file.
3. If the filter does extremely much work, then create a new thread for it.
4. Only store packets which are useful!
5. Don't store more than 2000 packets, it reduces the overall performance.

7.2 PacketSource

In order to add a new PacketSource, a new PacketSource must implement the following interface

7.2.1 PacketSource Interface

7.2.2 Additional Information

This chapter contains additional information for PacketSource writers.

1. Always allocate Packets dynamically!
2. Try to use reduce synchronization effort, i.e. run the PacketSource in no other thread than the PacketProcessingQueue thread.
3. If multithreading is used, make sure that stopCapture unblocks the PacketProcessingQueue thread and the PacketSource thread.
4. Notify the PacketProcessingQueue thread when an error happens.

5. Allow the PacketProcessingQueue consume all packets even when the capture has already stopped.

8 Attacking WEP

This section describes the different possible WEP-Attacks in YAWCU. A detailed description of WEP, Basic and Improved attack can be found in the FSM article. WEP is a stream cipher and uses the RC4-algorithm. This algorithm leaks information about the key. YAWCU uses this leaks to start an heuristically attack to crack the key. It works for 64 and 128 bit WEP - keys. If a key is found, YAWCU checks this key. Therefore it takes the packet, decrypt it, calculate its checksum and compares it with the append checksum of the packet.

8.1 Basic WEP Attack

The first attack, which YAWCU supports, is the standard FMS attack. It works as follows: YAWCU captures data packets. Every packet is sent to the method attack, which decides if the packet is saved or not. If not it is discarded. The first position of the initialization has to have (it has to have the value of the keybyte we are looking for) the correct initialization vector value in order that the attack takes place. The packet is processed and the result of the equation

$$z - j + S[i] = K[B]$$

is calculated. Then the value at position counts[K[B]] is incremented by one. After enough packets have been processed (the number is set by the user with the resolved variable in the config file) YAWCU evaluates the counts array. The element of the array which has the highest value is taken and set as correct keybyte. After this is done, YAWCU does the same with the remaining keybytes. When the last keybyte is found it checks the checksum of a message. It decrypts the payload, calculates its checksum and compares it with the checksum inside the packet. If it is incorrect YAWCU increases the number of resolved cases per keybyte about the value of the variable steprange (also defined in the config file). In this case all previous captured packets, saved inside the packetmanagment are used to find the key.

8.2 Improved WEP Attack

The improved WEP attack does the same as the basic WEP attack. But it has two improvements:

- Special resolved cases: This can happen if under the three important values a duplicate value occurs (the three values are S[1], S[S[1]] and S[S[1] + S[S[1]]]). If this happens, the possibility to guess a correct key byte increases from 5 % to 13 %. The

improvements are implemented in the function Weight. If a special resolved case occurs, our method returns true and 3 is added to the element in Counts instead of 1.

- Guessing early keybytes: The second improvement works as follows: Normally YAWCU waits with calculating a keybyte until a specific number of packets with the correct initialization vector (the first position of the IV has to have the same value as the position (exact: position - 3) we want to find in the key) has been reached. Early keybyte guessing doesn't wait. It gets all packets of every position in the key from the packetmanagment and tries to find the key. What does this mean? If B = 1 and our keylength is 5 we have already found the first and the second position of the key. We also have 20 captured packets for keybyte 3, 2 for keybyte 4 and 16 for keybyte 5. The user has specified the value resolved with 25. The third position of the key will not be calculated until 25 resolved packets for keybyte 3 are captured. But early keybyte guessing tries after every processed packet to find the key. This means YAWCU asks its packetmanagment for the 20 packets of position 3 and calculates the keybyte. Then it initializes the KSA with the new key and uses the two packets of position 4 to find this keybyte. This will be done until the key is found. Then the key is checked. If the key is found the program writes a statistic out else it restores the state before early keybyte guessing was done and continues as normal.

8.3 KoreK WEP Attack

The third possible choice for an attack is the KoreK attack. The difference to the basic / improved attack is, that the KoreK attack can use packets, where the initialization vector is in "resolved" form and packets which differ from this form. Shortly spoken, it can use all packets. Because of this circumstance it needed another packetmanagment which is explained in 8.4.2. The attack itself consists of 17 different attacks. 16 of them are attacking methods. It is possible to split them up in three groups:

- stable: basic and improved attack and some others
- semi-stable : means that the result of the attacks fails more often than the result of the stable attack
- unstable: means that the result of this attack fails more often than the results of the semi-stable attacks

The semi-stable and unstable methods cause that the key is found faster but it can be happen that he will never be found. Because of this we suggest to use only the stability mode zero or one. The last attacking method is an inverse attack. This means it is used to reject found wrong keybytes.
The attack itself works very simple. YAWCU waits until a specific number of packets is captured. Then it launches the attack. It takes every packet, attacks it with the different attacks. The result is an array votes, which contains the number of successfully attacks

of an specific attack for every possible ASCII sign . An method finds out which keybyte is the right one.After this is done, it takes the captured packets and does the same again until the length of the key is reached. Then the checksum tells us, if the key is correct. If YAWCU had found a wrong key there exist two possibilities:

- It tries the fudge improvement to find the key (see 8.5.2)
- It increases the number of packets to look at. YAWCU will make this automatically.

8.4 Packetmanagment in YAWCU

Generally the packetmanagment is needed because of several reasons. If YAWCU finds out that a key is wrong it has to reevaluate all keybytes. The packetmanagment secures that no necessary packet get lost. YAWCU implements two different forms of packetmanagment. They are explained below.

8.4.1 Basic, Improved attack

The basic and improved attack share the same packetmanagment. This packetmanagment just saves packets where the initialization vector is in a resolved form. The packetmanagment consists out of a vector with the size of the keylength. Every position of the vector contains a list. When a packet is added to the packetmanagment, YAWCU looks at the first value of the initialization vector and pushes a pointer of the packet on the list which belongs to the right position in the vector. YAWCU simply sorts all packets after its first position in the initialization vector and adds it to the corresponding list.

8.4.2 KoreK

The KoreK attack needs a lot of packets to attack a WEP key successfully. But it needs less packets than the basic / improved attack because it is possible to use nearly all packets for an attack. For the KoreK attack, YAWCU doesn't use the packetmanagment of the basic / improved attack. Instead it has implemented its own. This is very simply made. In the init function of the class KoreK we put five values in an array. This values are the initialization vector and the first and second byte of the payload. The two bytes of the payload are xored with the SNAP-header so that we have the scramble value z. Then a pointer pointing to the array is put on a list. After adding the array to the list, the original packet is discarded.If we are calculating a keybyte we iterate through this list and use every value to find the correct keybyte.

8.5 Improvements

YAWCU has implemented some improvements. Main goal of the improvements is to find the key earlier and less packets need to be captured by YAWCU. But the improvements, especially radius / fudge can slow down the computer dramatically. For this reason use the values suggested in 8.6..

8.5.1 ASCII Checker

This improvement was very easy to implement. It just checks, if the found keybyte is a number or letter. If not, the keybyte gets discarded. The ASCII Checker should not be used during live capturing and cracking. If a keybyte of the key is no letter or number, the key will never be found. It's always better to capture some pcap files (YAWCU is able to process more than one capture file) and crack them afterward.

8.5.2 Radius / Fudge / Breakradius

Basic, Improved:

All attacks which YAWCU performs are heuristic attacks. This means that YAWCU has an array in which all occurrences of every possible sign is counted. The index of the array (= sign) which has the highest value is chosen to be the correct one. It happens that an incorrect value is chosen out of several reasons. The radius / fudge improvement has the challenge to try all possible letters which are in radius and fudge. What does this mean? The following example shows it:

counts[49] = 25
counts[18] = 28
counts[50] = 30
counts[12] = 8

We suggest that the correct keybyte would be in counts[49]. But YAWCU would take the keybyte in counts[50] because it has the highest value. With radius / fudge we have the possibilities to tell YAWCU that he also should try the possibility which are in radius. If the radius would be 2, all possibilities which have a higher value than 50 percent of 30 will be taken and tried out. This can be a very high number. For this reason we can set the fudge. e.g.: YAWCU didn't find the correct key. So it uses radius and fudge to find it. Fudge has the value 1 and radius 2. This means all values above except counts[12] would be in radius. Since fudge is just 1, it would only try counts[18] and will not find the correct keybyte. If we would change the value of fudge into 2, it will try the the value of counts[18] first, and after it has recognized that this value is incorrect, it will take counts[49] and find out, that this keybyte is the correct one.
But radius / fudge don't only check the last keybyte. It starts at the end of the key and walks down until the beginning the key. Let us suppose that the total length of the key is n. If it finds out that the key doesn't match and so it goes down to the position n - 1. Then it recalculate the heuristic array and tries out all keys. If it replaces one keybyte in the key, it has to recalculate the value for the keybyte n. E.g.: There exist 5 resolved cases for every keybyte. The keylength is five. The key was not found so radius tries the position n - 1, in our case four. After fudge has the value of three, it looks if it finds three values in radius. Afterward (we assume that three values were found) it replaces the keybyte of n - 1 with one of the three. Than it has to reevaluate the position of n. The key was not found with this keybyte. So it takes the next byte of the three, replaces the keybyte and

reevaluates the position of the keybyte n. We assume that the keybyte was not found with this three values. So we have to look at the position n - 2, in our case 3. This goes on until we are at position 0 or we found the key (in this case the program prints the statistics and terminates). As this needs a lot of computation time YAWCU has possibility to set an earlier end to this recursion. This is implemented with the configuration value breakrad. It has a value between 1 and 12. This means that the recursion is stopped if it reaches the position n which is identical with the value of breakrad.

KoreK:

The KoreK attack consists out of 17 different attacks. For this reason the radius / fudge improvement is differently implemented. First there doesn't exist any radius anymore. The second difference is, that the program doesn't recalculate the following keybytes any more (e.g.: if n - 2 is changed, n - 1 is not recalculated), after one is changed. The reason is the huge amount of packets, which YAWCU would have to process. This improvement saves as many possible keybytes per keybyte as the the variable fudge defines: e.g.: if fudge is 3 it takes the the three values with the highest values in the result array (the result array is an array which contains for every character a number of how probable is this character to be the correct keybyte). After the key was found and incorrect YAWCU tries to find the key with this improvement. For this it takes the last keybyte and tries out the two other saved possibilities for the key. If it was false it goes down one position to n - 1 and replaces the keybyte there with one of the previous saved. The correct key is found if the checksum of the message, decrypted with the found key, and the one inside the captured message are equal. This improvement takes time and it is better to set the variable breakrad to limit the number of recursions YAWCU has to perform (more information in 8.6 or above (Basic/Improved)).

8.5.3 Guess Last Keybyte

This improvements simply tries out all possibilities for the last keybyte. For live capturing it is very useful because if you have all keybytes except the last one, it would be faster to try all 256 combinations than to wait for enough resolved cases to start an attack. Another advantage is that it definitely finds the last correct keybyte if the rest of the key is correct.

8.6 Arguments in the config file

- **keylength:** This value contains the length of the key. It can be 5 for an 64 bit key or 13 for a 128 bit key. It is necessary to set this value.
- **resolved:** This value has two different meanings.
  - Basic / Improved: The value specifies how many packets in resolved form per keybyte shall be waited before the attack on this keybyte starts. We suggest a value of 50.

  - KoreK: The value specifies how many packets shall be waited before the attack starts. We suggest a value of 200000.

- **radius:** This value contains the radius for the radius / fudge improvement. It is only necessary for the basic or improved attack. We suggest that this value should be 2. Information about this value can be found in 8.5.2.

- **fudge:** This value contains the fudge for the basic /improved and KoreK attack. If this value is high it slows down the speed of your computer dramatically. We suggest that this value should be less than 8. Information about this value can be found in 8.5.2.

- **breakrad:** This value is needed to cancel the recursion of the radius / fudge improvement. It can lie between 1 and 11 for a 128 bit key. We suggest a value of 7 for an 128 bit key and zero for a 64 bit key.

- **guess / last:** It is the same. A value greater zero enables the guess last keybyte feature. For Basic and Improved attack, it is the last value, for KoreK the guess value.

- **ascii:** A value greater zero enables the ASCII improvement.

- **steprange:** The steprange contains the value of how much the number of resolved are incremented when a key is not found with the current number of resolved cases.

- **stability:** With this argument it is possible to decide which stability mode shall be used for the KoreK WEP attack. We suggest to use the value zero or one.

- **snap:** It is possible to change the value of the SNAP - header with this variable. This variable should not be used. It was just for testing. The default value is 170.

8.6.1 Examples

The first example shows a sample configuration file for the basic attack.

```
<source>
<sourcename>WpcapSource</sourcename>
<sourcesymbol>wpcap</sourcesymbol>
<sourceconf>caps\64bitDump.pcap</sourceconf>
<sourcelimit>22223232323</sourcelimit>
<sourcemodel>single</sourcemodel>
</source>
<filter>
<filtername>BasicWEPFilter</filtername>
<filtersymbol>basicwep</filtersymbol>
<filterconf>keylength=5resolved=5radius=0fudge=0last=0steprange=5ascii=0</filterconf>
```

```
<filtermodel>single</filtermodel>
<filtersource>wpcap</filtersource>
</filter>
```

The second example shows a sample configuration file for the improved attack. The different to the configuration file above is that this one uses two pcap files.

```
<source>
<sourcename>WpcapSource</sourcename>
<sourcesymbol>wpcap</sourcesymbol>
<sourceconf>caps\korek.pcap;caps\all.cap</sourceconf>
<sourcelimit>22223232323</sourcelimit>
<sourcemodel>single</sourcemodel>
</source>
<filter>
<filtername>ImprovedWEPFilter</filtername>
<filtersymbol>basicwep</filtersymbol>
<filterconf>keylength=5resolved=15snap=170radius=2fudge=2last=1
steprange=5ascii=0breakrad=0</filterconf>
<filtermodel>single</filtermodel>
<filtersource>wpcap</filtersource>
</filter>
```

The third example shows a sample configuration file for the korek attack. This configuration file captures the data (it doesn't use a pcap file) live.

```
<source>
<sourcename>PeekSource</sourcename>
<sourcesymbol>peek</sourcesymbol>
<sourceconf></sourceconf>
<sourcelimit>22223232323</sourcelimit>
<sourcemodel>single</sourcemodel>
</source>
<filter>
<filtername>KorekWEPFilter</filtername>
<filtersymbol>korekwep</filtersymbol>
<filterconf>keylength=5resolved=200000guess=1ascii=0stability=1
fudge=0steprange=15000breakrad=0</filterconf>
<filtermodel>single</filtermodel>
<filtersource>wpcap</filtersource>
</filter>
```

9 Source Description

This section gives a short overview of the classes which are used for attacking WEP encryption.

9.1 RC4

This class is needed to verify the found key and to decrypt captured pcap files. The most important methods of this class are:

- **encrypt:** This method is used to encrypt files. It takes three arguments and has no return value (call by reference): a message array, the key and the length of the message. After the method has finished, the message is encrypted.
- **decrypt:** The decrypt method is used by the attack class. It just calls up the encrypt method because encryption and decryption do exactly the same.

9.2 CRC-32

This class is used to build the checksum of a message.

- **calculateChecksum:** This method takes three arguments and has no return value (call by reference). The checksum is saved in the third argument.

9.3 Attack

This class is a superclass for all attacking - classes. It contains all members and methods which are used by them. The most important members are:

- **B:** This value contains the position of the keybyte YAWCU is looking for.
- **counts:** This member is an array with the size of 256. Every index is the position of an character of the ASCII code.
- packetmanagment, rc4, crc32: The class attacks contains some object-members so that all derived classes can work with them.
- resolved, keylength, step, ... : All values which are specified in the config file are saved here.

The most important methods are:

- **attack:** This method is called by the basic and improved attack with every packet. It is used to decide if a packet is used for attack, to start an attack and to build the statistic.

- **basicIV:** This method checks if the initialization vector of a packet is in a resolved state. If not, it returns false and YAWCU discards the packet.
- **initKSA:** Most attacks need to initialize the key scheduling algorithms as long as possible. This method makes this and saves the important value j in ksa_j.
- **evaluateAttack:** This method evaluates the attack after enough resolved cases have been processed. In that case it looks if the key is found (if B == keylength). If not it increments B. Than it gets all packets of the packetmanagment which can be used to find the next position of the key and evaluates them.
- **findMax:**This method is used to find the correct keybyte in the array counts.
- **restart:** This method restarts the attack and increases the number of the resolved cases per keybyte about the value of step. It is called if the correct key is not found. It clears all necessary values (B, key_, ..), get all previous captured packets and tries to find the key again.
- **guessLast:** This method simply tries out all 256 possibilities for the last keybyte. If the right key is found (it calls checkChecksum) it returns true else false.
- **checkChecksum:** This method takes an packet, decrypt it with the previous found key, checks the checksum of the message and returns true if it is identically.

Every class, which wants to use this class has to implement the following virtual methods:

- **startAttack:** This method takes as argument a packet and returns one if the key has been found. It calculates a result and increase the value of this character in the counts array.
- **improvements:** This method is called in the method evaluate attack. Here it is possible to enable / disable different improvements (e.g.: early key byte guessing) for different attacks.
- **tryRadius:** This method is called inside the startAttack method if the fudge / radius argument is set. It tries to find the key, after YAWCU has found one but it was incorrect (checksum false). An explanation of its working can be found in 8.5.2.

9.4 Basic

- **improvements:** The basic improvements contains only the possibility to guess the last key byte. The ASCII improvement is realized in the findMax method.

9.5 Improved

- **improvements:** This method starts two possible improvements: the early keybyte guessing and the last keybte guessing. Both are described below.

- **Weight:**This is one of the two improvements of the improved attack. This method checks, if there is a duplicate value between three values of the state array. The values are state_[1], state_[state_[1]] and state_[(state_[1] + state_[state_[1]])%N]. A positive return value causes a higher value, which is added to the counts_ array.

- **guessKey:** The second improvement is early keybyte guessing. Normally YAWCU expects a specific number of packets (the value of resolved) before it evaluates the counts_ array. Early keybyte guessing does something different. If the key is not found until the moment, the method is called it tries to find the last positions of the key. For this it looks until which position the key is already found. Then it takes all packets which are in the packetmanagment and tries to find the other positions of the key. After it has found the last position the found key is checked with checkChecksum.

9.6 KoreK

- **initVariable:** This method initializes all important variables. It gets a packetpointer as argument. Out of this packet it builds an array and adds this on a list.

- **performAttack:** This method starts the attack. It calls one attack method after the other. There exist three stability modes. Which one is chosen depends on the value of the stability variable in the config file.

- **evaluateAttack:** This method has the job to evaluate the votes array and find the correct keybyte. It also saves its nearest neighbors for the fudge improvement.

- **restartKoreK:** This method is called by evaluateAttack. It has the job to perform the attack for every position B in the key except the first one. What does this mean? The KoreK attack uses the same packets to calculate every keybyte. The restart method is called after the first keybyte is evaluated. It increments B (contains the value which keybyte is calculated next) and gets one packet after the other and performs the attack on it. After YAWCU has finished this it calls evaluateAttack which is choosing the hopefully correct keybyte. Then evaluateAttack calls restartAttack again and that is going on until the end of the key. If the key is found the program terminates else it increases the specific number of packets to look at.

- **votes:** Votes is a member of the KoreK attack. It is needed to find the key. Votes is an array of arrays. The first array has the size of the number of attacks. The second contains the maximum number of characters, in our case 256. If a specific attack finds a keybyte it increase the value of votes at his position. (e.g: attack 5 finds keybyte 10, so the following will happen: votes[5][10]++)

- **attackXX:** There exist seventeen different attacks. 16 of them are used to find the key. Number seventeen has the challenge to reject false keybytes. The attacks always contains out of an condition. If this condition is correct, a keybyte is calculated and the value of the keybyte in the array is incremented.
- **checkFudge:**The method evaluate adds a number of values to an array. This array is the fudge array and contains the values which have the highest appearance in the votes array. The method checkFudge tries out all possible combination of the key with the values of the fudge array.