Abstract

This text was written on 2025-11-30 by the author nesca4 over the course of 3 days in Russian, and was translated on the same day by ChatGPT into English. It is likely not difficult to find spelling mistakes in the original version of the text; furthermore, the text is written in the form of a treatise — all of this is due to the fact that I had to work on it in great haste. Nevertheless, I sincerely hope that even in this form, it will prove to be of some benefit.

lomaster 2025.

INITIVM

(I). Thus, although the development of nesca4 is no longer being carried out officially, I continue to receive questions concerning it; accordingly, I resolved to compose this very treatise, whose aim is to set forth certain information about nesca4.

In truth, I wrote nesca4 in my youth as a project to learn how to write code, and for that reason it was completely rewritten several times; nevertheless, the latest version remains not entirely finished. The first version of nesca4 was written sometime in the winter of 2023; in the spring of that same year its development began to be published in the Telegram channel «OldTeam»; on July 31 it was posted in the Telegram channel «Tochka sbora». It should be clarified: the author has no relation to the authors of nesca3, though he may, unbeknownst to himself, have crossed paths with them.

Apparently the nesca4 project now scarcely astonishes anyone, yet in its time — relative to my abilities and my youth — it was a wonder; I once even loved to boast that nmap required twenty-five years to achieve what took me one year (how true that is I will not presume to judge). Of course, one

might rewrite it yet again and make it very fine, but, to be frank, I am grown weary and wish to leave it be: let it stand as it is.

As for libncsnet (formerly ncsock) (ncs, n — nesca), this is a library that in time flowed out of nesca4, much like libpcap from tcpdump: it so happens that, with the passage of years, a program's code accrues many things that are proper to be extracted into a separate library.

Generally, the development of libncsnet proceeded more steadily than that of nesca4, but it too has long been abandoned. And although much of its code was originally taken from nmap and libdnet, and the code is rather uniform, the raw.h interface contained within it I still regard as a rather artful solution.

(II). Thus, if one summarizes what nesca4 can do now — and to be brief — the list is as follows:

i. Scan ports by methods: (TCP): syn, xmas, null, fin, psh, maimon, window, ack; (SCTP): init, cookie; (UDP): udp.

ii. Perform ping by methods: (TCP): syn, ack; (ICMP): echo, timestamp, info; and also: arp, udp.

iii. Brute-force and scan services: HTTP basic auth, FTP. **Previously** there was also sup-

port for: Hikvision, RVI, SSH, RTSP.

iv. Query/lookup data against a database (such identification functionality existed in nesca3).

v. Save results in HTML (as in nesca3).

vi. Accept as targets: CIDR4, DNS, IPv4, RANGE4.

Old version (prior to the last rewrite) which supports brute-forcing RVI and Hikvision cameras. But it is rather poor, and this manual does not suit it — at best only slightly.

(III). Before use, bear the following in mind:

i. nesca4 is fairly complicated to use, runs only on Linux, and is not intended for ordinary users.

ii. nesca4 is made for more «professional» tasks than nesca3. It is designed primarily for various port-scanning activities rather than for brute-force, although it does support brute-force.

iii. nesca4 resembles nmap in many respects: its «flags» differ, but usually only cosmetically.

iv. The only documentation for nesca4 until now is its «usage menu»: a fearsome wall of text with options like: `-pps <pps> set your pps holy shit abcd 1234`; it is displayed on an empty run of nesca4 or when run with the `-help` flag: `./nesca4 -help`. However, long

ago (see the commits on GitHub) there was a file in `resources/` named «do_not_read.txt», which contained something like documentation.

v. Do not expect it to be exactly the same as nesca3.

(IV). Before launching nesca4, it must be compiled, i.e. built from source.

1. For this you must have on your Linux system: g++, gcc, make, and git. If they are absent, you should install them — this is probably not difficult.

2. If all of the above are present, to compile you should run the following four commands in sequence in a Linux terminal:

```
cd ~

git clone --recurse-submodules --depth=1 \
        https://github.com/oldteamhost/nesca4

cd nesca4

./configure

make -j
```

3. If your computer cannot cope with the compilation, instead of `make -j` type simply: `make`.

If the compilation still stalls, then within the `nesca4` directory (into which you have entered with `cd nesca4`) run the following five commands:

```
./configure

cd libncsnet

make CFLAGS="-fPIC -DHAVE_CONFIG_H \
            -flto -I."

cd ..

make CFLAGS=""
```

4. In my tests, compilation completes successfully on Arch Linux and Ubuntu.

5. If compilation finished successfully, a file named `nesca4` will appear in the `nesca4` directory; to verify its presence you can run `ls`: this command will print all files in the current directory. To learn the current directory run `pwd`; it should be something like:

```
/home/<user>/nesca4
```

BASIS

(I). Thus, nesca4 is a console utility for obtaining various information about network hosts; scanning is the process of obtaining that information. Nesca4 aims to provide information about network hosts as quickly, as elegantly, as accurately, and as comprehensively as possible.

Any nesca4 session proceeds as follows: (i) start nesca4, passing it at launch some «input data», (ii) receive the results of its work until it terminates.

It is important to note: nesca4 will perform its work according to the «input data» we supplied at startup. It is by means of these «input data» that it is configured, scan targets are supplied, and so on. Different «input data» = different behavior of nesca4.

(II). Any invocation of nesca4 is performed by entering the following command in a terminal:

```
./nesca4 "input data"
```

> Technical clarification: nesca4 can function fully

only when run as root; the command above assumes you are the root user, but in most cases that is not so, which is why `sudo` must be added to the command:

```
sudo ./nesca4 "input data"
```

You may also simply run

```
su
```

after which all subsequent commands will automatically run with root privileges (so there will be no need to add `sudo`).

Henceforth, we will likewise omit `sudo`, as done above.

(III). The input data are of the following three types:

1. Scan targets — these are input data that represent hosts to be scanned; they may be passed separated by spaces in the formats IPv4, DNS, CIDR4, RANGE4:

```
./nesca4 google.com 104.237.160.0/19
./nesca4 77.88.55.88 youtube.com
./nesca4 104.237.160.0-104.237.191.255
```

2. Flags (or options, arguments) — these are input data that indicate something: for example, how the work should be performed, what to use, and so on; additionally, flags can also specify hosts to scan, but not directly as above — rather indirectly.

To distinguish flags from scan targets, they are indicated with a dash (`-`), for example:

```
./nesca4 -syn -pe -p 80 google.com
```

Among these three flags the flag (`-p 80`) stands out, since it has its own «parameter», in this case — 80.

In general, flags are of two kinds: (1) those that have a «parameter», and (2) those that do not have a «parameter». As seen above, a flag's «parameter» is specified by a space after the flag itself: (`-flag param`). It is important to note that a flag's «parameter» can be only one (which is why `google.com` above is not considered the parameter of the flag `-p`).

> However, a flag's «parameter» may also be

specified like this:

```
-p80 google.com
```

In this notation, 80 is also the parameter of the flag `-p`, and `google.com` still is not its parameter.

Many programs — but not nesca4 — distinguish between short (single-character) and long (two-or-more-character) flags; in those programs long flags are indicated with two dashes (`-`) and short with one (`-`):

```
--flag
-f
```

Nesca4, for the sake of simplicity, does not do this.

3. The standard configuration file — this is input data that is a configuration file and is passed (even if you do not see it) on every run of nesca4, unless another configuration file is specified with the `-cfg` flag (we will speak below about this flag and about configuration files). This file is located at: `resources/config/default.cfg`.

(IV). So, we have launched nesca4 passing it input data; as noted above, we then «receive the results of its work until it terminates.»

By default nesca4 prints results directly to the terminal, but it can also save them in HTML format similar to nesca3. First we discuss the former.

1. At the beginning nesca4 prints a «start line»: a line announcing the launch of nesca4; it contains the version, time, and date of the run:

```
Running NESCA4 (v20240926) time
        05:52:30 at 2025-11-29
```

2. After that line, nesca4 begins to print the results of its work (i.e., the main portion of the output begins).

Nesca4 prints all of its results in blocks: the result of scanning one host = one block; we will call such blocks «result blocks». Here is an example of two of them:

```
Report nesca4 for 142.250.74.46 ...,

ports '80/tcp/open/http(syn)'

Report nesca4 for 77.88.44.55 ...,

ports '80/tcp/open/http(syn)'
```

3. Finally, after printing all results and just before terminating, nesca4 prints an «end line»: a line announcing the completion of nesca4; it contains the total number of reachable hosts and the total runtime:

```
NESCA4 finished 2 up IPs (success)
                      in 201.60 ms
```

(V). Any result block emitted by nesca4 consists of a «header» and of zero or more «nodes»:

1. «header» — the single mandatory element of a result block; it is the heading of that block. In its most complete form it contains: the target's IP address, the target's MAC address, the target's DNS, the methods used on the target, and the target's response time (RTT):

```
Report nesca4 for 1.1.1.1 (dns.com)
        [43:2a:61:5b:89:71] '.eS' 8.51 ms
```

It may be not entirely clear what is meant by «the methods used on the target» (although this is not an important element):

```
'.eS'
```

Essentially this is a string of the form: `'.<methods>'`, where `<methods>` is a sequence of letters, each letter denoting a scanning or ping method used against that target. For example, if we enabled ACK & ICMP ping and ACK scanning, the string would look like: `.eaA`. The full mapping of letters, if needed, exists in the code in the function `strmethod()`.

As noted, the «header» may contain the target's MAC address; this occurs when it was obtained via an ARP ping.

It was also noted that DNS entries and response times (RTT) may be multiple; regarding the latter: one successful ping = one RTT.

2. «node» — an element of the result block that contains some piece of information about the block's target; it is not the «header», and it is not mandatory, although «nodes» contain the principal information such as open ports, login and password, services (HTTP, FTP), etc.

Each «node» has the format (`<info type> <info>`):

```
ports    '80/tcp/unfiltered/http(ack)'
http(title)     google
```

In the last example, `<info>` is — `google`, and <info type> is — `http(title)`.

The format of <info> can vary widely; in the case of ports it is rather involved:

```
format = '<port>','<port>',...,

port = <num>/<protocol>/
        <status>/<service>(<method>)

protocol = "tcp" | "sctp" | "udp" | "???"

status = "open" | "closed" | "filtered" |
        "error" | "open|filtered" |
        "unfiltered" | "???"

method = "syn" | "xmas" | "fin" | "ack" |
        "window" | "null" | "maimon" |
        "psh" | "init" | "cookie" |
        "udp" | "???"
```

The <info> format for brute-force results is as follows:

```
login:pass@
```

Strictly speaking, a brute-force result node and

a service node are the same thing: a «node» with an HTTP header, and a «node» with a login and password for an HTTP page — these are simply «nodes» of the HTTP service in this case.

Services are listed in the file resources/nesca-services; it is by using this database that nesca4 determines services.

The «node» format for database-based identification is:

```
db(<key>)     <info>
```

Here, `<key>` is what nesca4 found in the target's information that led it to conclude `<info>`. For example, if nesca4 found the word «login» in an HTTP redirect, it will deduce that this is an authentication page:

```
db(login)     auth
```


The identification database resides in resources/nesca-

database; if one takes the trouble to study it, it can be modified. It is rather decent, given that it supports regex.

Its format is roughly:

```
Default:
'<key>','<info>',0,<where looking?>;

Regex:
R'<key>','<info>',0,<where looking?>;

<where are we looking?> =
    FIND_ALL  -1
    FIND_HTTP_REDIRECT 0
    FIND_HTTP_TITLE  1
    FIND_HTTP_HTML  2
    FIND_MAC  3
    FIND_DNS  4
    FIND_IP   5
    FIND_FTP_HELLO  6
```

For example:

```
    'kek','is kek',0,1
```

The «node» format for services is as follows:

```
<service>(type) <info>
```

Here, `<service>` is the name of the service, such as `http` or `ftp`, and `<type>` is the actual type of information about this `<service>`.

(VI). Finally, to save results in HTML you may use the flag `-html <file>`, which accepts as its parameter the path (filename) of the HTML file:

```
-html myfile.html
```

Open this file with a browser; you can do this directly from the terminal with the `firefox` command (if it is installed):

```
firefox myfile.html
```

Or with Google Chrome:

```
google-chrome myfile.html
google-chrome-stable myfile.html
```

If you have read the chapter above, the HTML save format will be familiar to you.

> In fact, the CSS style for the HTML files that nesca4 generates is located in resources/style.css.

(VII). Besides configuration via flags, nesca4 may be configured through configuration files. Essentially, this is the same configuration as via flags, but stored in a file and with a slightly different format.

1. We already spoke of the standard configuration file above; note only that, besides the default settings, this file contains a small guide to creating configuration files; we omit that guide here.

2. To specify another configuration file you may use the flag `-cfg file`, which accepts as its parameter the path to the configuration file to include:

```
-cfg resources/config/insane.cfg
```

When you specify your own configuration file, nesca4 ceases to automatically use the standard configuration file.

NVCLEVS

(I). Essentially, the entire operation of nesca4 consists of the following: (i) obtain scan targets, (ii) scan them, (iii) produce results, (iv) either repeat i–iii, or — if all targets have been scanned and results produced — terminate.

I ask: how does nesca4 scan the targets given to it? I answer: for each received target nesca4 performs 6 scan stages sequentially.

These 6 stages are: (1) ping scanning, (2) obtaining DNS names, (3) port scanning, (4) service scanning, (5) identification via the database, (6) brute-force.

As a target passes through these 6 stages, we gradually obtain information about it, and information obtained at one stage may be used by nesca4 at another stage: for example, the target's response time (RTT) is obtained by nesca4 at the first stage, and then used at the fourth stage to calculate timeouts. True: this is an extended scheme of nmap's operation.

For nesca4 to output anything, at least the first, second, or third stage is required; these can be called the main stages.

However, although the first and second stages

are enabled by default, their methods are not selected by default, so de facto they are still disabled. The same situation applies to the fourth stage: it is also enabled by default, but de facto disabled, since the ports for its operation are not specified by default.

370 It is important to understand: some stages may depend on each other, and therefore may sometimes be present or sometimes absent: for example, if no target passes the first stage, the others will not even start.

The current scan stage is displayed in the status bar at the bottom:

```
/ PASSED 44    PORTS SCANNING
            2.55 KiB  206.00 B
```

However, some stages execute so quickly that you may not even see them in the status bar.

The value `passed` is how many targets have been scanned.

The other two values are the amount of traffic received on your network interface and sent from your network interface. It simply reads the corresponding files from `/sys/class/net/`.

Some flags (all of which take no parameters) al-

low disabling these stages entirely: the flag `-n-ping` — disables the first stage, the flag `-n` — the second, the flag `-sn` — the third, the flag `-n-db` — the fifth, the flag `-n-brute` — the sixth. There is no flag for the fourth stage.

(II). Now we will give example commands and then explain them.

1. I give:

```
./nesca4 -html out.html -syn \
        -pe -p 80,443 google.com
```

I explain: such a call to nesca4 will give us the following information about our single target `google.com`: (1) the status of TCP ports 80 and 443, (2) response time, (3) DNS, (4) its IP address, (5) identification via the database.

The flag `-html out.html` is obvious, since we already discussed HTML saving.

The flag `-syn` enables the SYN method for port scanning; thanks to this the third stage will be operational.

The flag `-pe` enables the ICMP echo method for ping scanning; thanks to this the first stage will be operational.

The flag `-p 80,443` sets the ports to scan, namely: 80 and 443 both TCP.

The command above is written a bit oddly: firstly, I moved part of it to another line; secondly, on the previous line I placed a \.

The first I did so that it would fit on the page, and the second so that it would work in the terminal if you copy and paste it.

The terminal (more precisely, the shell running in it) requires that if you break a command onto the next line, you put a \ on the previous line, otherwise it will think the command on the next line is not a continuation but a new command.

That is, if I simply write,

```
./nesca4 -html out.html -syn
    -pe -p 80,443 google.com
```

the shell will think these are two commands. But if you put the \ as above, it will understand that this is one command.

In addition to simply listing ports separated

by commas, you can specify a port range using `-p`:

```
-p 80-85
```

The above entry is equivalent to this:

```
-p 80,81,82,83,84,85
```

2. I give:

```
./nesca4 -html out.html -n -syn \
        -s http:80,8000,8080,8888 \
        -pe -p 80,8000,8080,8888 \
        -random-ip 40000
```

I explain: this invocation of nesca4 is already fairly extensive: firstly, we are scanning 40000 random IPv4 addresses; secondly, we obtain much more information, since due to the flag `-s` we made the fourth stage operational; thirdly, we specified many more ports.

All information about a target that we can obtain with such a call is: (1) status of TCP ports 80, 8000, 8080, 8888, (2) response time, (3) IPv4 address, (4) HTTP response, (5) HTTP redirection, (6) HTTP login and password from the page, (7) identification via the database, (8) HTTP header.

Such scanning is excellent for finding various «hidden sites», because the fact that a host has port 80, 8000, 8080, or 8888 open almost always indicates an HTTP page (i.e., a website). Also, database identification plays a large role here: thanks to it one can understand which page is «typical» and which is something else. Other information may also be useful for this.

The flag `-n` disables the second stage of scanning; in general, DNS is not the most necessary thing, but obtaining it takes a lot of time, so this flag can significantly speed up the scan.

The flag `-random-ip 40000` instructs nesca4 to use 400000 random IPv4 addresses as scan targets; it is important to note: this does not prevent you from specifying other targets.

The flag `-s http:80,8000,8080,8888` tells nesca4 that ports 80, 8000, 8080, and 8888 should be treated as ports hosting an HTTP service; as a result, the fourth stage becomes operational because it now has ports and a service to work with.

By default, nesca4 truncates very long service

«nodes», for example a «node» with an HTTP response. If you want to disable this, there is a flag:

```
-detal
```

3. I give:

```
./nesca4 -html out.html -n -syn -pe \
        -pa 21,80 \
        -s http:80,8000,8080,8888,ftp:21 \
        -p 80,8000,8080,8888,21 -v \
        -import ip.txt
```

I explain: this is similar to the command above, but now we take scan targets from the file `ip.txt` (you may put DNS names, IPv4 addresses, CIDR4, RANGE4 there), we obtain information about FTP, use one more ping-scanning method, and emit verbose logs.

All information about a target that we can obtain with such a call is: (1) status of TCP ports 80, 8000, 8080, 8888, 21, (2) response time, (3) IPv4 address, (4) HTTP response, (5) HTTP redirection, (6) HTTP page login and password, (7) identification via the database, (8) HTTP header, (9) FTP banner, (10) FTP server login and password.

The flag `-import ip.txt` — tells nesca4 to take scan targets from the file `ip.txt`.

The flag `-v` — enables display of debugging information; this will turn nesca4's terminal output more into logs than result output; however it is expected that you view results in `out.html`.

The flag `-pa 21,80` — enables the TCP ACK method of ping scanning (previously we only used ICMP echo), and instructs it to use ports 21 and 80.

4. I give:

```
./nesca4 -html out.html -n -syn -pe \
        -pa 21,80 -login l.txt -pass p.txt \
        -s http:80,8000,8080,8888,ftp:21 \
        -p 80,8000,8080,8888,21 -v \
        -onlyopen -import ip.txt
```

I explain: Exactly the same as the command above, but now usernames and passwords for brute-force are taken from your files, and we output results only for targets with open ports.

The flag `-onlyopen` — instructs nesca4 to output and save only those targets that have open ports. Very useful to discard «noise».

The flag `-login l.txt` — tells nesca4 to take usernames for brute-force from the file `l.txt`.

The flag `-pass p.txt` — tells nesca4 to take passwords for brute-force from the file `p.txt`.

> By default, nesca4 takes usernames and passwords from the files:
>
> ```
> resources/login.txt
> resources/pass.txt
> ```

(III). Now let's talk about how to increase scan speed and what affects it. It is important to note: all these methods not only increase speed but also reduce scan accuracy. The methods are as follows:

1. As already mentioned above, you can increase scan speed by disabling the second stage, which is not that useful; this can be done with the flag `-n` (it takes no parameter). Important to note: disabling the first stage, not the second, may, on the contrary, slow the scan down.

2. Next, you can increase scan speed by using as few ping-scanning and port-scanning methods as possible. In practice, the combination `-pe -syn` is usually sufficient.

3. Next, you can increase scan speed by lowering the packet wait timeout for the first stage: the thing is, if the timeout for the third stage nesca4

calculates (which wildly increases speed), then for the first stage the timeout is always fixed at the start, because we don't yet have the data to calculate it. By default this is 800 milliseconds.

To lower it, you can use the flag `-wait-ping`, which takes as a parameter the new timeout value for the first stage. For example:

```
-wait-ping 500ms
```

In general, nesca4 accepts time values in these formats:

`10000`	`1000 nanoseconds`
`500ms`	`500 milliseconds`
`1s`	`1 second`
`10m`	`10 minutes`
`5h`	`1 hour`

4. Next, you can increase scan speed by changing group parameters.

The thing is, nesca4, like nmap, divides all targets you supplied into groups and scans them group by group. The working principle is: nesca4 receives 1000 targets; since the minimum group size is 100 (i.e. `gmin=100`), nesca4 forms a group of 100 targets and scans that entire group; then, because the group increment is 100 (i.e. `gplus=100`),

nesca4 increases the current group size by 100 (was 100, becomes 200); nesca4 will continue to increase the group size each iteration like this until the maximum is reached (i.e. `gmax`, by default 800).

This scheme is needed so that we can obtain scan results while the scan is still running, rather than waiting for it to finish completely. It is also useful to avoid filling all RAM by loading, for example, 1 million addresses at once.

So, you can change group parameters with these flags:

```
-gmax 3000
-gplus 300
-gmin 350
```

The most influential are the maximum and minimum group sizes, since the latter is the size of the very first group.

5. Next, you can increase scan speed by raising the maximum number of open file descriptors (sockets).

In general, to send and receive packets (and thus perform scanning) you need to open a socket through which these operations are performed. In nesca4 one socket = one thread, so the more sockets we open, the more scans we can perform concurrently. Important to consider: typically Linux

supports a maximum of 1024 open sockets, and some may be used by other applications, so you usually cannot set all 1024. Thus we do:

```
-maxfds 1010
```

This should significantly speed up scanning; however, keep in mind that sockets must still be closed, and that takes time. My solution to this problem in nesca4 is very bad, possibly even laughable, but it works: sockets in threads. I'm fairly sure this generates memory leaks and various errors, but the speed achieved is indeed high.

6. Next, you can increase scan speed by decreasing the timeout multiplier used for port scanning.

The thing is, as mentioned, when nesca4 scans ports it doesn't use a static timeout but a calculated packet timeout. It calculates this timeout based on the response time obtained in the first stage; that is precisely why disabling the first stage can only slow the scan. The calculation uses the formula: rtt * mtpl-scan = timeout, where `mtpl-scan` is the multiplier, default: 10.

The lower the timeout, the faster the scan; therefore, lowering `mtpl-scan` reduces the timeout. Do this like so:

```
-mtpl-scan 7
```

However, I do not recommend doing this — usually it is unnecessary.

(IV). Now let's talk about how to increase scan accuracy (and, among other things, bypass various protections, firewalls, etc.). It is very important to note: all the methods in the previous chapter, while they increase speed, if «opposite» will increase accuracy. For example, instead of `-maxfds 1010`, specify `-maxfds 10`. In general, in almost all cases the following formula holds: the higher the accuracy, the lower the speed, and vice versa. Because of this, we will have to repeat ourselves a little in this chapter:

1. Increase accuracy by raising the number of ping scans and port scans. By default nesca4 performs them once per host, but you can increase this value:

```
-num-scan 10
-num-ping 10
```

2. The more methods you use, the greater the chance that a method will work on a host. For TCP port scanning the methods and their flags are:

```
-syn -xmas -fin -null
-maimon -psh -null
```

However, all except `-syn` trigger very rarely, and some do not even allow determining whether a port is open, though `-fin` sometimes works.

For ping scanning the methods are:

```
-pe -pa80 -ps80 -py80 -pu53 -pi -pm
```

I listed them in order from most frequently to least frequently effective; I did not include ARP ping (`-pr`) because it only works on local hosts. Clarification: the `-pu` method is a UDP method, and `-py` is an SCTP method. If you wish, you can specify other ports instead of using my scheme with 80 and 53.

For SCTP port scanning the methods are:

```
-init -cookie
```

For UDP there is:

```
-udp
```

Important: if you want to scan SCTP or UDP

ports (which is very specific and rarely needed), you must specify ports for them:

```
-p S:80,443
-p U:53
```

Here the letter `U` means UDP, and `S` means SCTP; for TCP the letter may be omitted as we did above, or you may specify `T`:

```
-p 80,443
-p T:80,443
```

If you specify several, do it like this:

```
-p 80,443,U:53,S:443
```

To quickly enable all port-scanning and ping methods you can use the flags:

```
-all-ping
-all-scan
```

3. The other parameters we already covered:

```
-maxfds -gmin -gmax -gplus
```

```
-wait-ping -mtpl-scan
```

Simply specify them the «opposite» way from how we set them in the previous chapter: make `maxfds` smaller, `gmin` smaller, `gmax` smaller, `gplus` smaller, `wait-ping` larger, `mtpl-scan` larger.

4. Now regarding bypassing protections, which can also increase accuracy; they are usually bypassed by using different scanning methods, but there are also:

```
-dlen 100
-dhex 0xa
-dstr hellohost
```

All these flags allow you to add custom data to the packets that are sent during scanning and pinging; where possible, they will be added.

The first adds random data of size 100 (or whatever value you pass as the parameter) to the packet.

The second adds your data specified in hex (in this case `0xa`). Hex is hexadecimal notation — we will not explain it here.

The third adds your ASCII data to the packet, i.e. some string — in this case `hellohost`.

There are also the following flags (I'll list only the main ones):

```
-ipopt 0xa
-off df/mf/rf
-win 400
-adler32
```

The first allows you to set IPv4 options, it works the same way as `-dhex` with hex.

The second lets you set fragmentation flags; in the example above all three flags are specified, but you may specify any one of them. As you can see, two or more are separated by `/`.

The third lets you set the TCP window size; in our example 400.

The fourth tells nesca4 that the checksum for SCTP should be calculated not by the CRC32C method but by Adler-32.

(V). And so, let us now speak of all the flags that remained unmentioned earlier:

The flag `-dev ifname` — allows one to specify a custom network interface that will be used by to send and receive packets during scanning and ping operations. It accepts the name of the interface as its parameter. Available interface names may be discovered by executing the command `ip a`. By default, nesca4 selects the first operational interface.

The flag `-dst mac` — allows one to set a custom destination MAC address. By default, nesca4 resolves this address automatically for the chosen interface.

The flag `-src mac` — allows one to set a custom source (sender) MAC address. By default, nesca4 resolves this address automatically for the chosen interface.

The flag `-ip4 ipv4` — allows one to set a custom source IPv4 address. By default, nesca4 resolves this automatically for the chosen interface.

The flag `-ip6 ipv6` — allows one to set a custom source IPv6 address. However, this is futile, for nesca4 does not support IPv6 (though all structural elements for such functionality do exist internally).

The flag `-pps num` — allows one to cap the number of packets per second both for port scanning and ping scanning. By default, no limit is imposed.

The flag `-stats` — enables statistics for both port and ping scans. This option is activated automatically when the verbose flag `-v` is used, and may be considered its more lightweight form.

The flag `-ackn num` — allows one to manually set a custom TCP acknowledgment number used in both ping and port scanning packets.

The flag `-badsum` — instructs nesca4 to intentionally compute incorrect packet checksums when performing port or ping scans. This typically results in no response being received. Important to note: this does not affect the IP checksum.

The flag `-dbpath filepath` — allows one to specify a custom filesystem path to the identification database, used during the penultimate stage of the process.

The flag `-wait-brute time` — sets the packet timeout for brute-force operations. Its time format is identical to that of the `-wait-ping` flag.

The flag `-threads-brute num` — sets the thread count used during brute-force operations. While this may increase speed, it may also reduce accuracy.

The flag `-delay-brute time` — sets a delay interval between brute-force packets. This may decrease speed, but increase accuracy.