Key logging and browser hijacking spyware analysis

Diogo Matos
DETI
University of Aveiro
Aveiro, Portugal
dftm@ua.pt

Tiago Silvestre
DETI
University of Aveiro
Aveiro, Portugal
tiago.silvestre@ua.pt

David Araújo
DETI
University of Aveiro
Aveiro, Portugal
davidaraujo@ua.pt

João Almeida
DETI
University of Aveiro
Aveiro, Portugal
boia13@ua.pt

***Abstract*—In the last few years, spyware attacks have been frequently used by hackers to steal information from victims. This report presents the development and analysis of spyware targeting key logging and Firefox browser hijacking within Debian-based systems. The spyware's functionalities include capturing keystrokes, accessing browser-stored cookies, and decrypting stored passwords. All information is sent to an attacker through Discord using Discord webhooks. Static and dynamic analyses were performed, revealing the behavior and structure of the spyware. Both analyses helped us create a program that successfully detects (identifies the process ID of the running spyware) and is able to kill the process. This detection is based on a signature-based detection approach. Additionally, two attack scenarios were created to analyze the impact and how the spyware could be used in a real-world scenario.**

***Keywords*—Malware, spyware, key logger, browser hijacking, malware analysis.**

I. Introduction

Spyware is the name given to the class of software that is surreptitiously installed on a user's computer monitors a user's activity and reports back to a third party on that behavior [1]. In the work presented in this report, we explore and analyze spyware developed by us targeting key logging and browser hijacking. These functionalities enable to an attacker unauthorized access to sensitive user information. To find the identity of malware, its fingerprint, and behavior, we performed both static and dynamic analysis. The static analysis analyses the codebase and structure of the spyware, unveiling its architecture, flow, and potential points of exploitation. Then, the dynamic analysis involved executing the spyware within controlled environments to observe its runtime behavior. Additionally, we give some attack examples to illustrate how the malware could be used by an attacker. To finish, we present different approaches to detect and kill the malware, including a tool implemented to do so using signatures.

II. Malware implementation

A. Goal

When a target computer is infected, the attacker shall have access to all keys pressed by the user, Firefox's stored cookies, login events synced with the key logger, and deciphered Firefox stored credentials. Enabling the attacker to have access to the target's private information. The malware runs on Linux Debian-based machines with a Firefox installation.

B. Implementation / tools

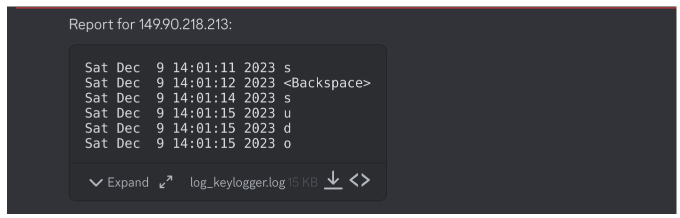


Fig. 1. Discord message with key log file for target with ip 149.90218.213.

The software is divided into three modules, the key logger, the browser data access, and the communication module. The key logger that we use is an open-source solution developed in C named Simple Key Logger [2]. To the code of the key logger was mode a minor change to add a timestamp to each entry in the log file, this is needed to be able to co-relate the key strokes with possible login events.

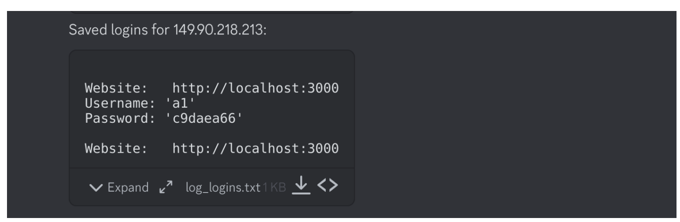


Fig. 2. Discord message with deciphered Firefox saved logins for target with ip 149.90218.213.

To interact with Firefox's history database we used a Python library called Browserexport [3]. The orchestration of the process of interaction with the browser was developed by us in Python. To access the cookies and possible login events, no extra tools were used. To decipher stored credentials we adapted an interactive tool named Firefox_decrypt [4] to a non-interactive version. That tool is based on a Firefox forensics tool named Dumpzilla [5]. Finally, the communication with the attacker is made using Discord webhooks, this way the attacker can see all data on a simple Discord server.

In the end, there are two distinct programs to be executed, the key logger and the browser hijacker/server (the Python side).

C. Example attacks

Some attack scenarios were created to illustrate potential exploits in real-life situations.

1) Scenario 1 - Teacher Attack: A professor sends an email to the students, instructing them to install the WINE software for the next class. Since WINE has multiple versions, the professor advises students to download WINE zip from a One Drive folder. The professor alters a seemingly harmless bash script within the WINE installation script, embedding the spyware. As WINE requires superuser permissions for installation, the spyware initializes upon execution.

Students, trusting in the professor, install WINE without verifying its version checksum and execute the spyware.

Then the professor steals cookies from students and gains access to varius systems used by them.

2) Scenario 2 - Github project poisoning: An attacker gains permission to modify a famous Github FOSS project that needs sudo permissions to be installed. He then makes a commit that inserts spyware into the project.

As a famous project, the repository is cloned very often and most users don't verify the correctness of the code. In the end, some users installed spyware without taking notice.

The attacker steals various information about the victims such as session cookies, passwords and logs. This information is then offered for sale online on the dark web.

III. Analysis

A. Static analysis

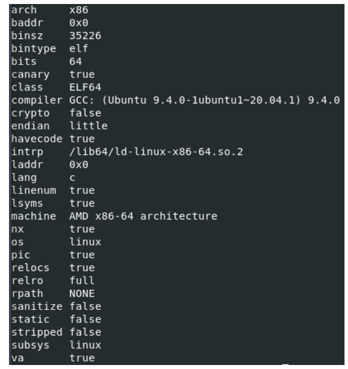
In this section, we present the knowledge about the malware that one can get by analyzing the code and produced files (e.g. binaries).

1) Key logger:

- To access the keyboard, the key logger first needs to detect the file where the Operating System (OS) is writing the typing events. To detect it, the program uses the fact that in *'/proc/bus/input/devices'* there's a complete description of all devices recognized by the OS. The resulting file will have the form *'/dev/input/*'*;
- The program opens the device using the file name previously discovered and the output file named *keylogger.log* located two directories back inside the *'log/'* directory where the logs are stored. Then, it reads the keyboard device file in an infinite loop that only stops if the device is disconnected. Every time it reads a new event, it checks if it is of type '1' and value '1' which means a key was pressed. Finally, it reads the code of the key pressed and stores it in the output log file;
- The process is daemonized and needs root permissions to execute.

Analyzing the binaries is also of vital importance. The following procedures were implemented.

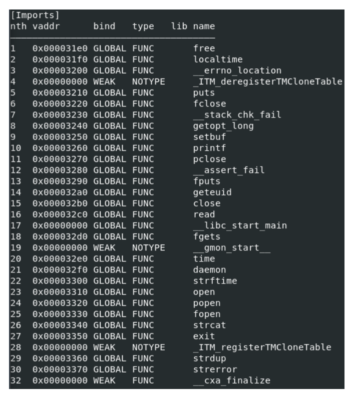
- File type identification using signatures. This was done using the *TrID*, a tool designed to identify file types from their binary signatures.
- The next was to search for any type of recognized structure in the binaries. For these, we can run a *YARA* tool in combination with a large set of *YARA* rules. No patterns were found within the rules.
- Performing a binary analysis with `Radare2` we can discover some metadata regarding the executable. By executing the command `rabin2 -I ./skeylogger` we get the output shown in Fig. 1.



```
arch     x86
baddr    0x0
binsz    35226
bintype  elf
bits     64
canary   true
class    ELF64
compiler GCC: (Ubuntu 9.4.0-1ubuntu1~20.04.1) 9.4.0
crypto   false
endian   little
havecode true
intrp    /lib64/ld-linux-x86-64.so.2
laddr    0x0
lang     c
linenum  true
lsyms    true
machine  AMD x86-64 architecture
nx       true
os       linux
pic      true
relocs   true
relro    full
rpath    NONE
sanitize false
static   false
stripped false
subsys   linux
va       true
```

Fig. 3. Metadata from the *skeylogger* binary

- Following this, we can list what modules this binary imports, as this may provide additional insight into what it is capable of. Running the command `rabin2 -i ./skeylogger` we get the output shown in Fig. 2.



```
[Imports]
nth vaddr      bind   type   lib name
―――――――――――――――――――――――――――――――――――――
1   0x000031e0 GLOBAL FUNC       free
2   0x000031f0 GLOBAL FUNC       localtime
3   0x00003200 GLOBAL FUNC       __errno_location
4   0x00000000 WEAK   NOTYPE     _ITM_deregisterTMCloneTable
5   0x00003210 GLOBAL FUNC       puts
6   0x00003220 GLOBAL FUNC       fclose
7   0x00003230 GLOBAL FUNC       __stack_chk_fail
8   0x00003240 GLOBAL FUNC       getopt_long
9   0x00003250 GLOBAL FUNC       setbuf
10  0x00003260 GLOBAL FUNC       printf
11  0x00003270 GLOBAL FUNC       pclose
12  0x00003280 GLOBAL FUNC       __assert_fail
13  0x00003290 GLOBAL FUNC       fputs
14  0x000032a0 GLOBAL FUNC       geteuid
15  0x000032b0 GLOBAL FUNC       close
16  0x000032c0 GLOBAL FUNC       read
17  0x00000000 GLOBAL FUNC       __libc_start_main
18  0x000032d0 GLOBAL FUNC       fgets
19  0x00000000 WEAK   NOTYPE     __gmon_start__
20  0x000032e0 GLOBAL FUNC       time
21  0x000032f0 GLOBAL FUNC       daemon
22  0x00003300 GLOBAL FUNC       strftime
23  0x00003310 GLOBAL FUNC       open
24  0x00003320 GLOBAL FUNC       popen
25  0x00003330 GLOBAL FUNC       fopen
26  0x00003340 GLOBAL FUNC       strcat
27  0x00003350 GLOBAL FUNC       exit
28  0x00000000 WEAK   NOTYPE     _ITM_registerTMCloneTable
29  0x00003360 GLOBAL FUNC       strdup
30  0x00003370 GLOBAL FUNC       strerror
32  0x00000000 WEAK   FUNC       __cxa_finalize
```

Fig. 4. *skeylogger* imported modules

- By examining the export, we can also see what function may exist. We can find one with the name of `getKeyText` which may suggest that the binary is collecting some sort of text.

2) Firefox data access:

- The first step made by the program is to find the location of Firefox's files. When installed using the *apt* package manager, the directory is *'/home/<username>/.mozilla/firefox/<profile>'*.
  In the same installation, multiple profiles might exist, but the one currently in use is defined inside *'/home/<username>/.mozilla/firefox/profiles.ini'*;

- To have access to history information, it connects to an SQLite3 database named *history.sqlite* inside the user profile directory. The Browserexport is used to access the DB and exports all data into a JSON format. Since we're dealing with SQLite3, there's no concurrent access to the database, so to export the history from it, Firefox needs to be shut down. That is not a problem because the browser will not be always open and the export is made very fast (about one second);
- To get the stored cookies, the program simply connects to the database *cookies.sqlite* and selects the name, and value columns from the table *moz_cookies*. From this information, we can know all cookies and domains of the website associated with them;
- To have access to the user credentials stored by the browser locally is not so trivial. The credentials are stored in a JSON file named *logins.json*. In this file a list of stored logins can be found, where the usernames and the passwords are encrypted, only the inherent website name is in plaintext. Of course, the keys used to encrypt the credentials are also stored locally, since Firefox needs to have access to them every time the user wishes the use any stored credentials. Firefox uses a cryptography interface named *PK11SDR* based on PKCS #11 (Public-Key Cryptography Standards #11). The Python script calls the C method *PK11SDR_Decrypt* to decipher the credentials. Using this interface the program can get the plain text credentials through a high-level call. This operation might require a username and password to unlock PK11SDR but, by default, there's no need.

3) Communication:

- It sends periodic messages, text, and text files, to three distinct webhooks (key logger, logins, and cookies);
- All communications are made to 'https://discord.com/api/webhooks/'.

B. Dynamic analysis

Dynamic analysis is the analysis of the properties of a running program. While dynamic analysis cannot prove that a program satisfies a particular property, it can detect violations of properties as well as provide useful information to programmers about the behavior of their programs [6]. We're especially interested in the last one, letting us understand better how the malware behaves and enabling the future development of detection tools.

To conduct a dynamic analysis of a running program, we utilized the `sysdig` tool. This tool allowed real-time monitoring of system activities, providing detailed insights into the operating system's behavior. Specifically, we focused on monitoring the `write` and `read` system calls related to a Python (`python3`) program.

1) Utilized `htop` to identify the `Process ID (PID)` associated with the `main.py` script.
2) Employed the obtained PID in `sysdig` for detailed system call analysis.

`sysdig` *Output: Read and Write Operations*

The `sysdig` output, depicted in Figure 5, provided valuable insights into the dynamic behavior of the Python program. As expected, the analysis revealed frequent read-and-write operations, indicating that the program was actively reading and writing data.

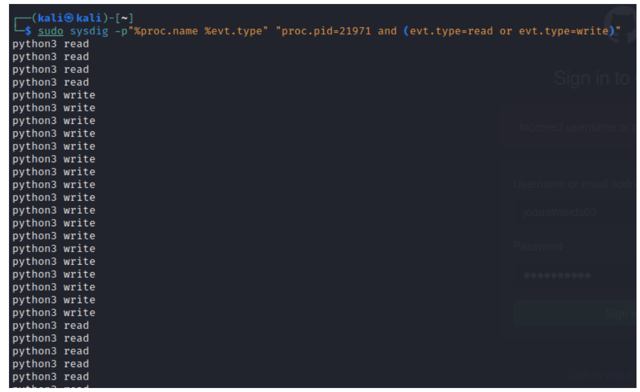


Fig. 5. `sysdig` output Showing Read and Write Operations

PID of `main.py`

In addition to `sysdig` output, a figure (Figure 6) illustrating the `PID` associated with `main.py` was included. The PID identification was facilitated using `htop`, allowing for the specific identification and tracking of the main process of the Python program.

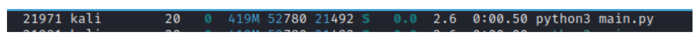


Fig. 6. PID of `main.py` Identified with `htop`

The `skeylogger` binary generates a process, which can be found by names, as well as all the files this process interacts with. To access this information, we can use the command `lsof -i -c skeylogger` which results in the output seen in Figure 7.

Using `sysdig` we can filter for the process name and the type of event it is creating. For the `skeylogger`, we know that it must read from the keyboard and write the captured data to some log file, so we can filter for these also. This will produce the output in Figure 8.

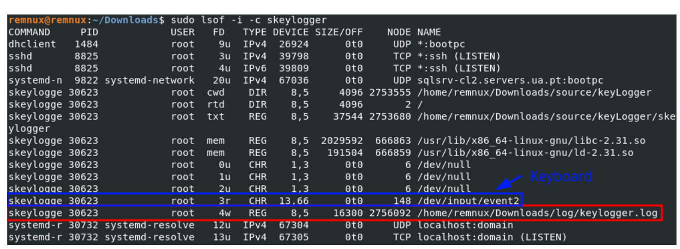


Fig. 7. Process interactions

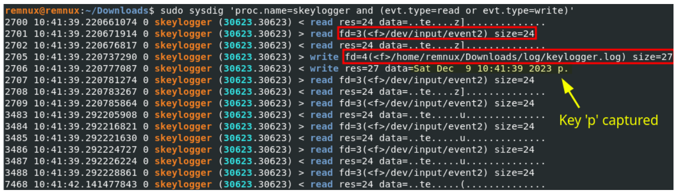


Fig. 8. Key capture and log processes

IV. Detection

A. Approaches

There are multiple ways to detect the malware when running. Usually, they're divided into three distinct categories [7]:

- *Signature based* - Based on the detection of a bit sequence in the files part of the malware, known as the signature. Note that signature-based detection methods are good at detecting known malware but unable to detect unknown malware and polymorphic malware because they can change their signatures. But it still is the most common approach to detect malware and is used by all antivirus programs today;
- *Heuristic based* - Aims to detect uncommon behavior in the computational system. Usually, this involves training an artificial intelligence model in an unsupervised fashion with a large normal behavior dataset. This approach has some limitations, especially the level of false positives and high data dependency;
- *Specification based* - In this technique, the knowledge about the malware is used to detect common patterns performed by the malware in a running system. This includes analyzing, for example, communications and file access. With this approach, threats can be detected even unknown malware. One of the major drawbacks of this technique is that, on average, it will much longer to detect the malware.

B. Signature-based detection tool

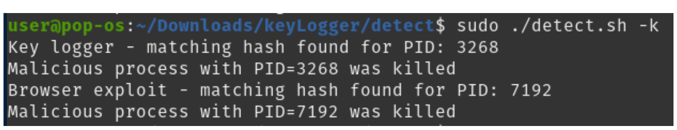


Fig. 9. Malware detection script (*detect.sh*) output when the malware is running.

We decided to use a signature-based approach in a simple tool developed in Bash that analyzes all running processes. For each, it calculates the signature of the executable behind the process and compares it to the pre-calculated hash of the two files that the malware needs to run.

The full path of the executable attached to a given process can be retrieved by reading this file *'/proc/<pid>/exe'*, where the *<pid>* is the process in analysis. We use SHA256 as a digest function, so the hash is calculated using the command *sha256sum*. Since the Python side of the malware, the Firefox exploit, and the server, are not compiled, the executable is the Python's binary that is always the same for every Python program running. To find the proper Python file, all Python files inside */proc/<pid>/cwd/* are analyzed. Note that in the *'/proc/<pid>/cwd/'* directory there is a symbolic link to all files that are inside the directory where the program was started.

The developed tool can detect or, if desired, kill the malicious processes.

V. Conclusion

In this project a spyware was developed and some attack scenarios were created to illustrate how the spyware can be used in a real scenario.

Then a static analysis was made which explains how the spyware works internally, a dynamic analysis was also performed. Both of these analysis gave us insights of how to create a tool to detect the spyware.

In the end, we identified some approaches that can be used to detect our spyware. A signature-based detection approach solution was developed to successfully detect and stop the spyware.

References

[1] Egele, Manuel, et al. "Dynamic spyware analysis." (2007).
[2] Singh, Gulshan. Simple key logger. GitHub. URL: https://github.com/gsingh93/simple-key-logger/
[3] Browserexport. GitHub. URL: https://github.com/seanbreckenridge/browserexport
[4] Alves, Renato. Firefox decrypt. GitHub. URL: https://github.com/unode/firefox_decrypt
[5] Dumpzilla. URL: www.dumpzilla.org
[6] Ball, Thoms. "The concept of dynamic analysis." ACM SIGSOFT Software Engineering Notes 24.6 (1999): 216-234.
[7] Idika, Nwokedi, and Aditya P. Mathur. "A survey of malware detection techniques." Purdue University 48.2 (2007): 32-46.