Syslogk Rootkit. Executing Bots via "Magic Packets"

David Álvarez-Pérez[1]
[1]Avast Software s.r.o. (Gen Digital Inc.) Pikrtova 1737/1a, 140 00 Praha 4, Czech Republic.

This paper was presented at Botconf 2023, Strasbourg, 11-14 April 2023, www.botconf.eu
It is published in the Journal on Cybercrime & Digital Investigations by CECyF, https://journal.cecyf.fr/ojs

Abstract

The proliferation of open source Linux kernel rootkits allows malware writers to speed up the process of developing complex malware. This study analyzes the Syslogk Linux kernel rootkit family which reuses code of Adore-Ng. Syslogk allows to remotely execute arbitrary commands and a hidden bot in different modes via "magic packets". Analyzing "magic packets" in reasonable time is a challenging task. Furthermore, the hidden bot, implements a proxy mode that allows to hide the IP address of the attacker while executing commands in other infected machines. This new botnet structure can also inspire future Linux threats, including IoT threats.

Keywords: syslogk, rootkit, botnet, netfilter.

1 Introduction

In early 2022, Jan Neduchal and I, discovered Syslogk Linux kernel rootkit hiding a backdoor identified as Rekoobe. Such backdoor, instead of being continuously running in the system, is remotely executed by the attacker via "magic packets"[1].

Later, based on a Vladimír Žalud rootkit detection, in October 2022, I identified a new variant of Syslogk[2] hiding a bot[3] that can implement multiple fake services (see Table 2). In this second version of Syslogk, the bot supports a proxy mode (see Table 1) and stealth communication messages faking Mozilla Firefox and Apache 2 (see Appx. 3 p. 46 making harder to identify the attacker and exposing a new botnet architecture based on "magic packets".

The present paper analyzes Syslogk rootkit family and addresses the problem of analyzing "magic packets" in reasonable time. The rest of the paper is structured as follows.

- Description of the problem and possible solutions
- The implementation experimented and its evaluation
- Discussion on the results
- Related work
- Future work
- Conclusion

2 Problem and possible solutions

Open source malware allows malware writers to speed up the development process[4][5][6][7]. It is very frequent to see code reuse from one Linux kernel rootkit to others, specially in those cases where the code snippets are complex to write or it's hard to find an alternative equivalent code.

At Avast, we experienced an increase in the number of rootkits that registers NetFilter hooks[8] for implementing "magic packets"[9].

The problem is that, in some cases, it is complex to determine the actions that an attacker can trigger remotely via "magic packets" in reasonable time.

For the current existing malware, it is quite simple to identify the basic blocks that implements the actions executed by the "magic packets". In practice, those typically execute actions in user mode space via call_usermodehelper[10][11] kernel API. So, the basic blocks containing cross-references to it are usually the target of the "magic packets". To put it differently, those calls are a side effect in the NetFilter hooks.

The analysis of "magic packets" is a similar problem to other more researched: determining if an unsafe function is causing a vulnerability in the program. Since, in both cases, the analyst must to determine the partition of the input space that is able to trigger the special case.[12] The main difference between both problems is that "magic packets" are more likely to be implemented to complicate the analysis. In fact, the attacker can implement highly data-flow sensitive "magic packets". For instance, during the "magic packet" parsing, Sysglok checks some internal variables whose value depends on other "magic packets"[1][13].

So, when the analyst identifies a basic block that is likely to be triggered by "magic packets", should answer, basically, the following three questions.

- Can be, this candidate basic block, reached via "magic packets"?
- Are the "magic packets" sufficient for reaching it?
- Which restrictions must to satisfy the "magic packets" for reaching the basic block?

For solving the problem of analyzing magic packets, I explored the following options:

- Develop a magic packet fuzzer.
- Deploy a Linux kernel debugging environment.
- Trace the NetFilter hooks using Kprobes.

Fuzzers has demonstrated being very effective finding vulnerabilities. But when analyzing "magic packets" we can assume a challenging scenario for fuzzing. For instance, Syslogk performs comparisons against encrypted values. The constrains generated by such comparisons will be infeasible to solve using a theorem prover[14] (i.ex. break AES encryption). Therefore, fuzzing can be helpful assisting the analysis, but a fully automatic solution can be not feasible[15][16].

Deploying a debugging environment is a manual solution that can aid the static analysis of the "magic packets". But deploying a kernel debugging environment[17] requires time and doesn't produce test cases easy to reproduce by others.

As discussed next, I decided to fully analyze the "magic packets" by tracing the rootkit.

2.1 Proposed solution on analyzing magic packets

Fortunately, modern Linux operating systems incorporates a Kernel tracing facility called Kprobes[18][19].

This facility allows to develop a kernel driver for tracing the value comparisons against the packet and dynamically printing these traces. This way, the analyst can produce two programs: a "magic packet" sender[13] and a driver[20] for tracing the packets. This way, apart of the documentation, the analyst generates a proof of concept (PoC) code that others can easily check without having too much knowledge on the specific rootkit. Apart from that, those Kprobes can be also used for analyzing other malware campaigns that simply changes the keys, and so on.

An additional problem is that this kind of rootkits tend to use some techniques for protecting themselves, (for instance, Syslogk removes itself from the list of loaded modules maintained by the operating system)[21]. This kind of protections are usually easy to defeat by patching the program. I provide a Python script for disabling such protection in Syslogk rootkit.

2.2 Figures and tables

```
[root@centos7 Kprobe_debug_Syslogk_Rootkit]# dmesg
[ 2576.405878] disksctracer_driver_cleanup : disksctracer module unloaded
[ 2583.258226] init_kprobe : Planted kprobe at ffffffffc0a0b875
[ 2583.259532] init_kprobe : Planted kprobe at ffffffffc0a0bdd1
[ 2583.260866] init_kprobe : Planted kprobe at ffffffffc0a0bf85
[ 2583.262049] init_kprobe : Planted kprobe at ffffffffc0a0b8a2
[ 2583.263342] init_kprobe : Planted kprobe at ffffffffc0a0b88b
[ 2583.268302] init_kprobe : Planted kprobe at ffffffffbb6baa00
[ 2583.268303] disksctracer_driver_init : disksctracer module loaded
[ 2663.807483] nfin_packet_fields_checks_handler_pre : magic packet protocol: [6]
[ 2663.807485] nfin_packet_fields_checks_handler_pre : The protocol fits the requirements
[ 2663.807486] nfin_packet_fields_checks_handler_pre : magic packet reserved: [2]
[ 2663.807487] nfin_packet_fields_checks_handler_pre : The reserved fits the requirements
[ 2663.807488] nfin_packet_fields_checks_handler_pre : magic packet data_offset_shifted_four_bits_rig
ht: [5]
[ 2663.807489] nfin_packet_fields_checks_handler_pre : Encrypted data expected to be at offset (shl d
ata_offset, 4) * 4 = 20
[ 2663.807500] nfin_packet_magic_check_handler_pre : RDI register. Data encrypted using EncodeDecode
[D3"jeoirtfbvs]
[ 2663.807501] nfin_packet_magic_check_handler_pre : ESI register. The magic value expected to be int
o the encryted data [11223344]
[ 2663.809819] call_usermodehelper_exec_handler_pre : handler_pre: Executing command in usermode spac
e from kernel [/bin/sh -c  echo command executed > /tmp/avast.txt]
[root@centos7 Kprobe_debug_Syslogk_Rootkit]#
```

Figure 1: Tracing command ID 99

Argument	Description
	The bot listens for one request and ends the execution.
cb	The bot enters in a loop receiving callback notifications for handling requests.
port	Fixes the port for the aforementioned commands.
proxy	Acts as a proxy for executing commands in other infected machines.

Table 1: Bot commands.

Protocol ID	Address	Description
0	0x00406286	Other protocol
1	0x00406275	tcp
99	0x00406299	ohttp
2	0x00405CAE	ssl
3	0x00405CCF	https
4	0x00405CF0	smtp

Table 2: Bot component. Supported protocols or fake services".

Variables	CMD 1	CMD 2
logininfo	Source address	Source address
logininfo+4	N/A	Source port
logininfo+6	Destination port	Destination port
Bot port	0	0
CMD status	1	3
state	normal	proxy

Table 3: Syslogk Rootkit. Internal variables.

Hook	PF	Hook Num	Priority
nfinpro	TCP	1	INT_MAX
nfout	TCP	3	INT_MIN
nfin	TCP	1	INT_MIN

Table 4: List of Syslogk rootkit NetFilter hooks.

ID	Command
0	/etc//tp-b8PbR2v1ms/sm1v2RbP8b cb
1	/bin/sh -c /etc//tp-b8PbR2v1ms/sm1v2RbP8b
2	etc//tp-b8PbR2v1ms/sm1v2RbP8b proxy
99	/bin/sh -c command_to_execute_here

Table 5: List of commands executed via "magic packets".

3 Implementation and evaluation

This study propose the following generic steps for efficiently analyzing "magic packets":

1. Identify the NetFilter hooks involved in network traffic packet analysis and/or modification.

I identified three NetFilter hooks:

- nfin. It is responsible of retrieving the port of the hidden bot.
- nfout. It is responsible of properly updating the outgoing packets (i.ex. proxy mode).
- nfinpro. It implements the magic packets parsing. If the packet fits the requirements, a Linux work executes the command in usermode space.

The NetFilter hooks definition allows to identify the input space and the candidate paths for reaching the actions triggered via "magic packets". In this case, the input space consist in IP/TCP packets Table 2 and all the candidate paths belongs to nfinpro NetFilter Hook. Notice that dependencies out of the scope of nfinpro can be systematically identified during the analysis.

2. Identify the basic blocks that implement actions candidate of being executed via "magic packets".

In the case of Syslogk rootkit, all the commands Table 2 are executed via call_usermodehelper API. By analyzing the cross-references to this API, it is straigthforward to determine that the function start_exec receives the Command ID in the esi registry of the processor which is hardcoded on each call to it. Also, all of those calls to start_exec take place into the nfinpro NetFilter hook so, this analysis reveals: the input space, the candidate paths, the constraints, and finally the target basic blocks containing calls to the usermode actions.

3. Remove anti-analysis protections from the malware.

Before inserting the Syslogk rootkit Linux kernel module for the analysis, it is convenient to patch the function that hides the module (offset 0xAB5). For instance, by replacing the first byte of the function by the x86 and x86_64 RET instruction (opcode 0xC3).

4. Build templates for the magic packets according to the supported protocols.

In the case of Syslogk, the "magic packets" consists in TCP/IP packets. The PF flag set to 2, in the NetFilter hooks, indicates Internet Protocol. While the comparison against 6 in the Protocol field of the IP packets (offsets 0x1821 and 0x0F4D) indicates TCP protocol. So, for implementing those "magic packets" the template consists in two arrays of 5 double-words. The first array for the IP header and the second one for the TCP header. The first nibble of the IP header indicates the IP version, for instance, 4. And the second one, the length of the IP header specified in double-words. 5 double-words in this case. Also, the byte at offset 9 (starting the count by zero), indicates the protocol. It

will be set to 6, in other words, it will be set to TCP protocol. The rest of the bytes will be initialized to NULL (0x00).

5. Extract the requirements for the "magic packets".

By statically analyzing the Syslogk nfinpro NetFilter hook, it is possible to determine the requirements for the "magic packets". The procedure consists in to edit the "magic packet" template with the appropriate values, extracted from the constraints found in the nfinpro function flow-graph. Each data added to the template is considered an hypothesis that must to be checked by tracing the module with Kprobes. The trace must print the values evaluated in the condition, and the path taken after such evaluation. Kprobes are also useful for printing values that are computed in memory by the rootkit. For instance, in the case of Syslogk rootkit, it is useful to print encrypted and then encoded values for adding those to the magic packet's template.

The result of the previous procedure consists in the following elements:

- Unprotected version of the rootkit.
- Script for sending "magic packets".
- Kprobes based Linux kernel module which dynamically traces the "magic packets" parsing.

3.1 Evaluation

By systematically applying the proposed method I extracted the "magic packets" associated to each command in reasonable time.

The template fixes the IP header, TCP header and data section of the packet as follows.

- Version and IHL: 0x45
- Protocol: 0x06
- Data Offset : 0x5

The Data Offset multiplied by 4 indicates the offset from the start of the TCP header to the data of the packet.

Next, the specific data for each command is discussed.

- Command id 99 Fig. 1 magic packet requirements:
  - Reserved: 0x02.
  - Packet data:
    * Magic value: 0x44332211
    * Key: 0x756a656f6972746662767300
    * 20 bytes of padding.
    * The Arbitrary bash command to execute.
    * Add padding, if necessary, for having, at least, 293 bytes of data.
    * As last step, it is required to flip the last bit for all the bytes in the data.
- Command id 2 magic packet requirements:
  - Running proxy mode:
    * Identification: Any from the whitelist (see Appx. 1 p. 45).
    * Reserved, Flags and Window Size: 0x500803FE
    * Data: "__step1__" encrypted and then encoded with the function FormatEncode (see Appx. 2 p. 46).
  - Updating values of logininfo Table 3:
    * The magic packet must to be sent just after running proxy mode.
    * Identification: Any from the whitelist (see Appx. 1 p. 45).
    * Reserved, Flags and Window Size: 0x500803FE
    * Data: "__step3__" encrypted and then encoded with the function FormatEncode (see Appx. 2 p. 46).
- Command id 1 magic packet requirements:
  - Without killing previous instances:
    * Identification: Any from the whitelist (see Appx. 1 p. 45).
    * Sequence Number: Any from the whitelist (see Appx. 1 p. 45)
    * Reserved: 0x0
    * Flags 0x02
    * Window Size: 0x03FE
  - Killing previous instances:
    * Identification: Any from the whitelist (see Appx. 1 p. 45).
    * Packet data:
      · Magic value: 0x0000002C
      · Key: "__step1__" encrypted and then encoded with the function
      · 6 bytes of padding.

Notice that commands with id 1 and 2 (used for starting the bot) sets the attacker logging information Table 2. Such logging information determines if the next magic packets received by Syslogk rootkit comes from the attacker or should be not taken into account. The nfout NetFilter hook Table 4 is responsible of propagating those values in Proxy Mode Table 1.

4 Discussion

Syslogk uses well-known Linux Kernel Rootkit techniques and even reuse code from other kernel rootkits (i.ex. Adore-Ng[7]).
Contrary to its predecessors, Syslogk integrates the usermode malicious component with the kernel module, implements fake services for connecting with the bot[1] and uses "magic packets" in a nobel proxy mode Table 1.
All these features and peculiarities can inspire other threat actors for developing stealth and adaptive botnets. A manual approach was sufficient for analyzing the Syslogk rootkit "magic packets" in reasonable time but, taking into account that the complexity of the "magic packet" parsing increases in new versions, it is necessary researching more on how to efficiently analyze "magic packets".

5 Related work

Linux kernel-level rootkits can be distributed in multiple forms depending on how those access to kernel space. The more frequent method is to distribute the rootkit as a Linux kernel module.[22] For modifying the behavior of the system, the rootkit can approach in two ways, giving place to the following classification: Kernel Object Hooking (KOH) rootkits, if it hijack kernel control flow, and Dynamic Kernel Object Manipulation (DKOM) rootkits, if it subvert the kernel by directly modifying dynamic data.[23] Adore-ng rootkit implements both methods. It removes itself from the module list by changing prev and next pointers of module-struct during Adore-ng's initialization[21] but also hijack kernel hijacks the following functions.[24]

- The routines used by the Virtual File System (VFS) to intercept (and modify) calls that access files in both the /proc file system and the root file system.
- The function for appending the last TCP/IP version 4 connections to /proc/net/tcp "sequential" file.

Reptile is other LKM-based x86_64 kernel mode rootkit that includes an encrypted backdoor which can spawn a reverse shell upon receival of a magic packet[9] Syslogk reuses the code of Adore-Ng for hiding itself, the malicious files and the TCP connections from the host. It also implements "magic packets" but those are complex and integrates the rootkit with the malicious hidden bot. [25]

6 Future work

The proposed procedure allows to analyze the Syslogk "magic packets" in reasonable time and produces sufficient material for other researchers to reproduce the tests.

The main problem of this approach is that it is fully manual. More research is needed to provide a semi-automatic solution. I suggest exploring solutions based in symbolic execution[26], concolic execution[27] and fuzzing[12] to provide a semi-automatic approach.

7 Conclusion

Syslogk reuses code from Adore-Ng[7] for hidding the malicious artifacts. It also implements complex "magic packets" that can be time-consuming to analyze. This study provides the procedure for analyzing Syslogk "magic packets" in reasonable time by using Linux Kprobes[19][18][20]. This paper also analyzes the proxy mode of the bot hidden by Syslogk rootkit. This new feature of controlling bots via "magic packets", indirectly, using a proxy mode, can inspire other malicious actors for creating new botnet architectures.

Appendices

Appendix 1

Syslogk rootkit word and double-word values whitelists for the magic packets.

IP header Identification field whitelist:
0x27E5, 0x6CC8, 0xF575, 0xEAEC, 0xF24A, 0xF322, 0x5044, 0x85BF, 0x60F2, 0x4D19, 0xCEDA, 0xAD05, 0x332F, 0xADB6, 0x99BD, 0xD2D6, 0xF02D, 0xBB1D, 0x4DC0, 0x7F95, 0xE04B, 0x75A0, 0xB48F, 0x35F1, 0xB0B7, 0xC5D2, 0x3DBD, 0xFF15, 0x4218, 0x2BF7, 0xB304, 0x66CC, 0x8BC9, 0x9C4E, 0x5A7B, 0xDD7E, 0x3B14, 0x4ED0, 0xB518, 0xF07E, 0x2D0C, 0x58FC, 0x9966, 0x8989, 0x357F, 0xBFE8, 0x6C83, 0x39AC, 0xDF2A and 0x4AB3.

TCP header Sequence number field whitelist:
0x36DF0DBE, 0x2E850DA1, 0x31307614, 0x622E52A2, 0x90F411A4, 0x92D8543B, 0xDA53E996, 0xC8D4B32E, 0x221356CC, 0x28A5AC2C, 0x3174D34E, 0x000A23AA, 0xC64B5258, 0x6CBD8489, 0xA5804552, 0x5B0D383A, 0x0E2E5552, 0xE5C2B533, 0x161AA6C4, 0x582F9043, 0x67E4591A, 0x8D252200, 0x0E454D90, 0x949FBB70, 0x9A98542C, 0x58E7A911, 0x6D5B67A9, 0x8C6901F1, 0x32E42AB0, 0x8D9B8A6B, 0x136BE259, 0xD21A9B88, 0xE13D60D5, 0xD766EB63, 0xBD7D9E51, 0x8D65C47E, 0x158BF6F0, 0xBB96784E, 0x388A9A3E, 0x3F104C3, 0x78C296BB, 0x26707F61, 0xB6703DB8, 0x45A91E39, 0xE264B6C6, 0x71C6894E, 0x9A67E79E, 0x64ACC1CF, 0x4E85FFF8, 0xA1BB5068, 0x10C04997, 0xD27AAB8B, 0xD8B36A23, 0x8DEEE21A, 0x0295CE1D, 0x386D0E91, 0xD1F20EE9, 0xC788438B, 0x8C865CEB, 0xEB17B69E, 0x1C6968A9, 0x6277EF09, 0xB01C9C00, 0xD5865D0E, 0xD384344C, 0xD505F015, 0x05333FBD, 0x811298A3, 0x3584A50F, 0x5C40977A, 0x5549FB3E,

0xA5A54B01, 0x3CCABC0E, 0x3D990DFE, 0x62455526, 0x8AF90BE3, 0x2F9AC244, 0x892A6001, 0xD2CFF2C4, 0x77EE4C5B, 0x0C7B225A, 0x3F769119, 0x0B9652AB, 0x10B98C3B, 0x3CFB293D, 0x45AAA237, 0x8D837B67, 0x86785F52, 0xE1644B21, 0x3BB5DFD7, 0xDCDEA543, 0x61C9DEE3, 0x43AD90EA, 0x9CCE04DA, 0x3AB88DED, 0x2D0B16E7, 0xAA00A321, 0x276749B6, 0xE937E089 and 0x24F70247

Appendix 2

Encryption functions:

- SimpleEncodeDecode
  - Encryption algorithm: RC4
  - Label: rc4_key
  - IV: 12 A3 BB 47 53 5E C0 D5 39 53 A6 FB AD 43 F5 73
  - Key: 63 7C 77 7B F2 6B 6F C5 30 01 67 2B FE D7 AB 76 CA 82 C9 7D FA 59 47 F0 AD D4 A2 AF 9C A4 72 C0 B7 FD 93 26 36 3F F7 CC 34 A5 E5 F1 71 D8 B1 15 04 C7 23 C3 18 96 05 9A 07 12 80 E2 EB 27 B2 75 19 83 2C 1A 1B 6E 5A A0 52 3B D6 E3 29 E3 2F 84
- SimpleEncodeDecode_0
  - Encryption algorithm: RC4
  - Label: L7_rc4_key
  - IV: 12 A3 FE 47 93 5E 12 D5 39 53 22 FB BD 43 98 73
  - Key: 07 FD 36 26 2C 3F F7 CC 34 AB E5 71 51 08 01 15 63 7C F2 7B C9 6B 6F C5 30 09 67 2B 00 17 2B 76 1A 82 16 7D 0A 59 47 F0 AD DB A2 AF AC 14 72 20 19 83 12 1A 1B 6E 5A A0 52 37 D6 E3 19 13 2F 14 04 C7 55 13 18 96 05 9A 07 23 80 02 0B 27 32 75
- SimpleEncodeDecode_1
  - Encryption algorithm: RC4
  - Label: manager_rc4_key
  - IV: 19 03 12 DA 1E 6E E5 A0 69 53 82 BB BD F3 98 76
  - Key: 07 FD 36 26 2C 3F DD 3C 34 AB B5 A1 51 08 91 15 B7 BD 93 D6 F6 5F F7 CC 44 A5 C5 F1 71 D8 B1 F5 1A 82 16 ED 0A 59 2B 73 AE F3 25 7D 35 8B 72 20 19 03 12 DA 1E 6E E5 A0 52 37 46 E3 99 13 2F 14 04 C7 55 63 18 96 05 9A E7 23 80 02 0B 27 32 75
- EncodeDecode
  - Encryption algorithm: XOR
  - Label: xorkey
  - Key: 1101link
- EncodeDecode1
  - Encryption algorithm: XOR
  - Label: xorkey1
  - Key: d3i9szdn
- EncodeDecode2
  - Encryption algorithm: XOR
  - Label: xorkey2
  - Key: 40239jig
- EncodeDecode3
  - Encryption algorithm: XOR
  - Label: xorkey3
  - Key: n430jdfk
- EncodeDecode4
  - Encryption algorithm: XOR
  - Label: xorkey4
  - Key: vndia323
- EncodeDecode5
  - Encryption algorithm: XOR
  - Label: xorkey5
  - Key: dnj23fds
- FormatEncode and FormatDecode
  - Encryption algorithm: AES
  - Label: key
  - Mode: CTR (Counter mode)
  - IV: 12 A3 BB 47 53 5E C0 D5 39 53 A6 FB AD 43 F5 73 - Key: 60 3D EB 15 15 3A 71 5E 2B 73 AE F3 85 7D 75 8B 1F 55 2C 57 3E 61 58 D7 2D 98 11 A3 39 14 DE FE

Appendix 3

Simulation of legitimate traffic:

- Structure of the packet's data simulating Mozilla Firefox:
  Connection: keep-alive\r\n
  User-Agent: Mozilla/5.0 (X11; Linux x86_64; rv:52.0) Gecko/20100101 Firefox/52.0\r\n
  Accept: text/html,application/xhtml+xml, application/xml;q=0.9,*/*;q=0.8\r\n
  Accept-Encoding: gzip, deflate\r\n
  Connection: keep-alive\r\n
  Content-Type: application/x-www-form-urlencoded\r\n
  POST /index.html HTTP/1.1\r\n
  Host: **HOST_GOES_HERE**\r\n
  Content-Length: **CONTENT-LENGTH_GOES_HERE** \r\n\r\n
- Structure of the packet's data simulating Apache Tomcat:
  HTTP/1.1 200 OK\r\n
  Date: **THE_DATETIME_GOES_HERE** GMT\r\n
  Content-Length: **CONTENT-LENGTH_GOES_HERE** \r\n\r\n
  Connection: close\r\n
  Cache-Control: no-cache\r\n\r\n
- Structure of the packet's data hiding messages generated by the function FormatEncode:
  GET /index.html HTTP/1.1\r\n
  Host:host\r\n
  FORMAT_ENCODED_DATA \r\n
  Accept: text/html,application/xhtml+xml, application/xml;q=0.9,*/*\r\n
  Cookie: ID=
  Connection: keep-alive\r\n

```
Accept-Encoding: gzip, deflate\r\n
User-Agent: Mozilla/5.0 (X11; Linux x86_64; rv:52.0) Gecko/20100101 Firefox/52.0\r\n
\r\n\r\n
```

Acknowledgment: I want to thank my colleagues of Avast (Gen Digital Inc.). Especially the Avast Threat Labs team and, more specifically, the following people: Sigudur (Siggi) Stefnisson, Luis Corrons, Jiří Sejtko, Jakub Křoustek, Jan Širmer, Bohumír Fajt, Vladimír Žalud, Ondřej David and Jan Neduchal. Thank you for making this research possible. Thank you for your help and support.

Author details

David Álvarez-Pérez

Avast Software s.r.o. (Gen Digital Inc.)
Pikrtova 1737/1a, 140 00 Praha 4, Czech Republic
david.alvarez@avast.com

References

[1] "Linux Threat Hunting 'syslogk' a kernel rootkit found under development in the wild." Avast. Accessed: 2023-02-26.

[2] "Syslogk rootkit malware sample." VirusTotal. Accessed: 2023-02-26.

[3] "Syslogk bot malware sample." VirusTotal. Accessed: 2023-02-26.

[4] "Reptile linux kernel rootkit." https://github.com/f0rb1dd3n/Reptile. Accessed: 2023-02-26.

[5] "RKDuck linux kernel rootkit." https://github.com/QuokkaLight/rkduck. Accessed: 2023-02-26.

[6] "KoviD linux kernel rootkit." https://github.com/carloslack/KoviD. Accessed: 2023-02-26.

[7] "Adore-NG linux kernel rootkit." https://github.com/yaoyumeng/adore-ng. Accessed: 2023-02-26.

[8] R. Rosen and R. Rosen, "Netfilter," *Linux Kernel Networking: Implementation and Theory*, pp. 247–278, 2014.

[9] J. Junnila, "Effectiveness of linux rootkit detection tools," 2020.

[10] "Toy Linux kernel rootkit with basic keylogging and backdoor capabilities." https://github.com/soad003/rootkit/blob/master/rootkit.c#L136. Accessed: 2023-02-26.

[11] "Out-of-Sight-Out-of-Mind-Rootkit linux kernel rootkit." https://github.com/NinnOgTonic/Out-of-Sight-Out-of-Mind-Rootkit/blob/master/osom.c#L211. Accessed: 2023-02-26.

[12] X. Zhu, S. Wen, S. Camtepe, and Y. Xiang, "Fuzzing: a survey for roadmap," *ACM Computing Surveys (CSUR)*, vol. 54, no. 11s, pp. 1–36, 2022.

[13] "Syslogk research tools." https://github.com/avast/ioc/tree/master/SyslogkRootkit/Research%20ToolsAvast. Accessed: 2023-02-26.

[14] "Microsoft z3 theorem prover." https://github.com/Z3Prover/z3. Accessed: 2023-02-26.

[15] A. BOVE, A. KRAUSS, and M. SOZEAU, "Partiality and recursion in interactive theorem provers – an overview," *Mathematical Structures in Computer Science*, vol. 26, no. 1, p. 38–88, 2016.

[16] G. Sutcliffe and C. Suttner, "Evaluating general purpose automated theorem proving systems," *Artificial Intelligence*, vol. 131, no. 1, pp. 39–54, 2001.

[17] R. Love, *Linux Kernel Development: Linux Kernel Development _p3*. Pearson Education, 2010.

[18] "Kernel Probes documentation." https://docs.kernel.org/trace/kprobes.html. Accessed: 2023-02-26.

[19] R. Krishnakumar, "Kernel korner: kprobes-a kernel debugger," *Linux Journal*, vol. 2005, no. 133, p. 11, 2005.

[20] "Spotify KProbes examples linux kernel module." https://github.com/spotify/linux/tree/master/samples/kprobes. Accessed: 2023-02-26.

[21] J. Wang, P. Zhao, and H. Ma, "Hacs: A hypervisor-based access control strategy to protect security-critical kernel data," in *2nd International Conference on Computer Science and Technology (CST 2017). Guilin, China, DOI: https://doi.org/10.12783/dtcse/cst2017/12516*, 2017.

[22] C. Kruegel, W. Robertson, and G. Vigna, "Detecting kernel-level rootkits through binary analysis," in *20th Annual Computer Security Applications Conference*, pp. 91–100, 2004.

[23] Z. Wang, X. Jiang, W. Cui, and P. Ning, "Countering kernel rootkits with lightweight hook protection," in *Proceedings of the 16th ACM Conference on Computer and Communications Security*, CCS '09, (New York, NY, USA), p. 545–554, Association for Computing Machinery, 2009.

[24] C. Kruegel, W. Robertson, and G. Vigna, "Detecting kernel-level rootkits through binary analysis," in *20th Annual Computer Security Applications Conference*, pp. 91–100, IEEE, 2004.

[25] M. L. Bak, L. Buttyán, and D. F. Papp, "Tee-based remote platform attestation,"

[26] R. Baldoni, E. Coppa, D. C. D'elia, C. Demetrescu, and I. Finocchi, "A survey of symbolic execution techniques," *ACM Computing Surveys (CSUR)*, vol. 51, no. 3, pp. 1–39, 2018.

[27] K. Sen, "Concolic testing," in *Proceedings of the twenty-second IEEE/ACM international conference on Automated software engineering*, pp. 571–572, 2007.