Packer

Kiteshield Packer is Being Abused by Linux Cyber Threat Actors

Alex.Turing, Wang Hao

2024年5月28日 • 8 min read

Background

Over the past month, `XLab's CTIA(Cyber Threat Insight Analysis) System` has captured a batch of suspicious ELF files with low detection rates on VT and very similar characteristics. Eager to delve into this, we commenced reverse engineering, facing a series of `anti-debugging techniques, string obfuscation, XOR encryption, RC4 encryption`, and more. Honestly, this process was quite exhilarating for us; the more obstacles we encountered, the more exceptional the samples seemed, making us feel as though we had caught a significant target.

However, the outcome was disappointing. The low detection rates of these samples were due to the use of Kiteshield packing. Ultimately, we discovered that these samples were known threats belonging to the **APT group Winnti, the cybercrime group DarkMosquito, and an unknown script kiddie**.

Even though we did not uncover any new threats from this batch of samples, the low detection rate itself is a crucial finding. It is evident that cybercrime organizations of varying levels are starting to use Kiteshield for evasion, and security vendors currently lack sufficient awareness of this packer. Therefore, we believe it is necessary to release this article to share our findings with the community and improve antivirus engines' capabilities to handle Kiteshield packer.

What is Kiteshield Packer?

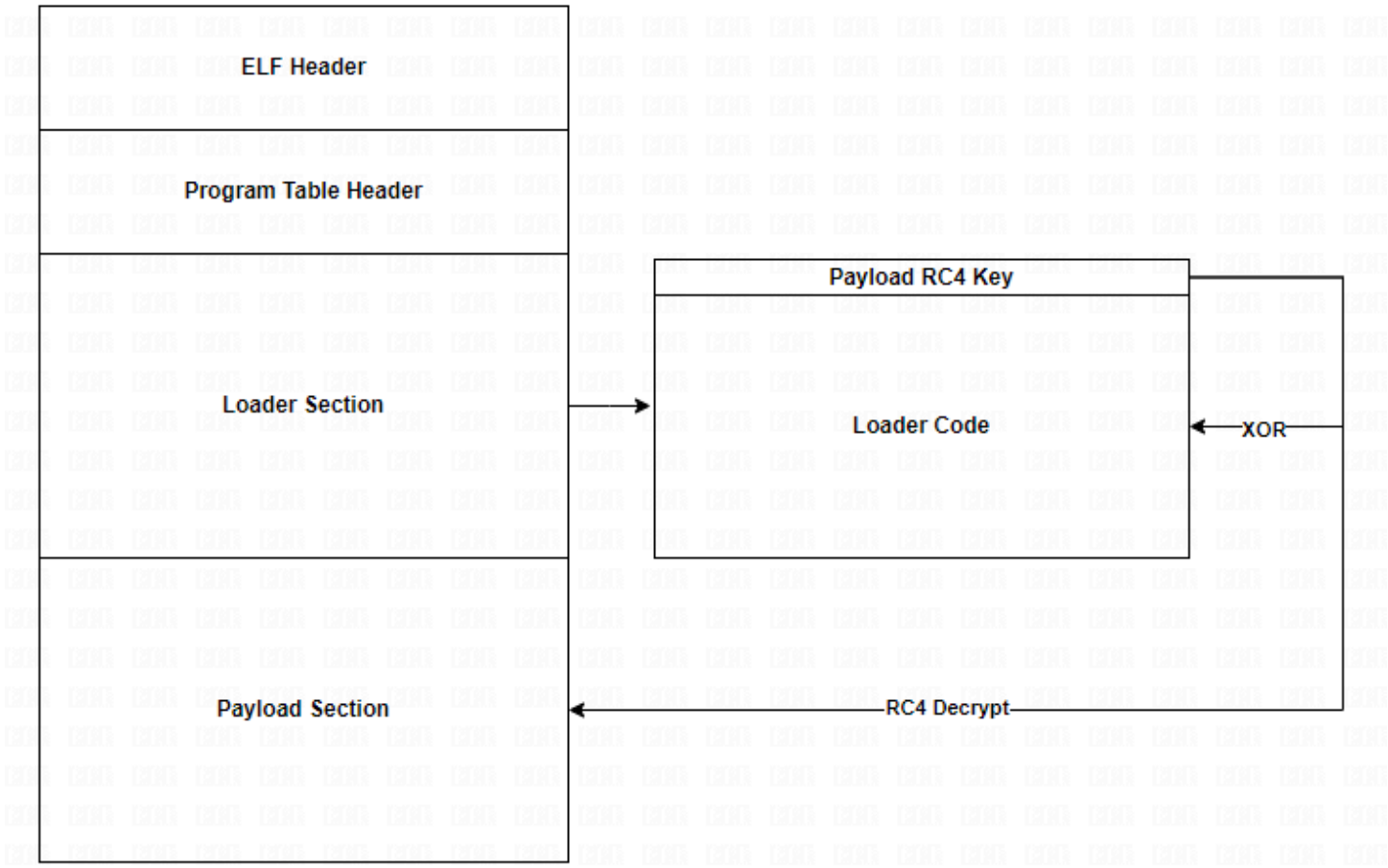
Kiteshield is a packer/protector for x86-64 ELF binaries on Linux. Kiteshield wraps ELF binaries with multiple layers of encryption and injects them with loader code that decrypts, maps, and executes the packed binary entirely in userspace. A ptrace-based runtime engine ensures that only functions in the current call stack are decrypted at any given time and additionally implements a variety of anti-debugging techniques in order to make packed binaries as hard to reverse-engineer as possible. Both single and multithreaded binaries are supported.

The following sections will focus on the characteristics of binaries packed with Kiteshield and how to unpack them. For more technical details on Kiteshield, readers can refer to its source code on GitHub.

First glimpse of Kiteshield

ELF files packed with Kiteshield contain only two segments: the first segment is the Loader section, and the second is the Payload section.

Their layout in the ELF file is as follows: the Loader section uses RC4 to decrypt the Payload.



The initial `rc4_key` is located between the end of the Program Table Header and the Entry Point, with a length of 16 bytes.

The `rc4_key` is randomly generated. For example, in `909c015d5602513a770508fa0b87bc6f`, the initial `rc4_key` would be `85 7F 6B A4 DD 39 5A A1 3E A7 A3 A8 11 77 E0 8E`.

The `rc4_key` cannot be used directly. First, it must be XORed with the loader code. This method ensures that the loader code has not been modified; otherwise, the correct RC4 key cannot be obtained. Afterward, the RC4 decryption takes place, and the decrypted payload is mapped into memory, followed by a jump to the new entry point.

The code that jumps to the new entry point, `jump to payload section`, originates from the following assembly snippet in the source code

`loader\entry.S`. This snippet is fixed and unchanging, serving as a characteristic signature of Kiteshield.

```
xor %edx, %edx
  xor %eax, %eax
  xor %ecx, %ecx
  xor %esi, %esi
  xor %edi, %edi
  xor %ebp, %ebp
  xor %r8d, %r8d
  xor %r9d, %r9d
  xor %r10d, %r10d
  xor %r11d, %r11d
  xor %r12d, %r12d
  xor %r13d, %r13d
  xor %r14d, %r14d
  xor %r15d, %r15d
  pop %rbx
  jmp *%rbx
```

Tricks in the Loader

0x1: Anti-Debugging

Kiteshield employs four anti-debugging techniques:

1. **antidebug_proc_check_traced**: Checks `/proc/<pid>/status` for the presence of the `TracerPid` field.
2. **antidebug_prctl_set_nondumpable**: Sets the process's dumpable flag to 0, preventing `ptrace` from attaching or dumping.
3. **antidebug_rlimit_set_zero_core**: Sets environment variables `LD_PRELOAD`, `LD_AUDIT`, and `LD_DEBUG` to empty values.
4. **antidebug_rlimit_set_zero_core**: Sets the `RLIMIT_CORE` (maximum core dump size) to 0 using `setrlimit`.

0x2: String Obfuscation

Kiteshield uses `DEOBF_STR` to implement single-byte XOR encryption, with the key hardcoded as `0x83`.

```
#define DEOBF_STR(str)                                                  \
  ({ volatile char cleartext[sizeof(str)];                              \
     for (int i = 0; i < sizeof(str); i++) {                            \
       cleartext[i] = str[i] ^ ((0x83 + i) % 256);                      \
     };                                                                 \
     cleartext[sizeof(cleartext) - 1] = '\\0';                          \
     (char *) cleartext; })
```

The following are the encrypted strings in the Loader, primarily used for anti-debugging.

```
STRINGS = {
    // loader/include/anti_debug.h
    'PROC_STATUS_FMT': '/proc/%d/status',
    'TRACERPID_PROC_FIELD': 'TracerPid:',

    // loader/runtime.c
    'PROC_STAT_FMT': '/proc/%d/stat',

    // loader/anti_debug.c
    'LD_PRELOAD': 'LD_PRELOAD',
    'LD_AUDIT': 'LD_AUDIT',
    'LD_DEBUG': 'LD_DEBUG',

    // loader/string.c
    'HEX_DIGITS': '0123456789abcdef'
}
```

In Kiteshield-packed files, the following code snippets, used for decrypting strings, are frequently encountered.

Based on the logic of `DEOBF_STR`, here's the corresponding Python code to achieve decryption. The string at address `0x2019E0` is decrypted to `/proc/%d/status`.

```
import idc
enctext = idc.get_bytes(0x00000000002019E0, 15)
plaintext = bytearray()
for idx, value in enumerate(enctext):
    plaintext.append(value ^ (0x83 + idx))
```

```
print(plaintext))
```

Detection & Unpacking

Based on the mentioned entry points and encrypted string characteristics, we implemented the following YARA rule and Python scripts for detection and unpacking.

Yara rule

```
import "elf"
rule kiteshield{

    strings:
        $loader_jmp = {31 D2 31 C0 31 C9 31 F6 31 FF 31 ED 45 31 C0 45 31 C9 45 31
        // "/proc/%d/status"
        $loader_s1 = {ac f4 f7 e9 e4 a7 ac ee a4 ff f9 ef fb e5 e2}
        // "TracerPid:"
        $loader_s2 = {d7 f6 e4 e5 e2 fa d9 e3 ef b6}
        // "/proc/%d/stat"
        $loader_s3 = {ac f4 f7 e9 e4 a7 ac ee a4 ff f9 ef fb}
        // "LD_PRELOAD"
        $loader_s4 = {cf c0 da d6 d5 cd c5 c5 ca c8}
        // "LD_AUDIT"
        $loader_s5 = {cf c0 da c7 d2 cc c0 de}
        // "LD_DEBUG"
        $loader_s6 = {cf c0 da c2 c2 ca dc cd}
        // "0123456789abcdef"
        $loader_s7 = {b3 b5 b7 b5 b3 bd bf bd b3 b5 ec ec ec f4 f4 f4}

    condition:
            $loader_jmp and all of ($loader_s*) and elf.type==elf.ET_EXEC and elf.m
}
```

Unpacking Script

```
import struct
import re
import lief
```

```
from Crypto.Cipher import ARC4

rt_info_pattern = rb".\x00\x00\x00.\x00\x00\x00.{8}[\x08-\x0a]\x09\x0a\x0b"
def rt_info_parser(data):
    nfuncs, ntraps = struct.unpack("<II", data[:8])
    # print(ntraps, nfuncs)
    rt_info_size = 17 * ntraps + 32 * nfuncs
    res = bytearray()
    for i, c in enumerate(data[8:8+rt_info_size]):
        res.append((c^i)&0xff)
    traps_start = res[:]
    trap_list = []
    for i in range(ntraps):
        traps = {}
        addr, ty_type, value, fcn_i = struct.unpack("<QIBI",traps_start[17*i:17*i+1
        # print(hex(addr), ty_type, value, fcn_i)
        traps['addr'] = addr
        traps['type'] = ty_type
        traps['value'] = value
        traps['fcn_i'] = fcn_i
        trap_list.append(traps)
    funcs_start = res[17*ntraps:]
    func_list = []
    for i in range(nfuncs):
        funcs = {}
        id, start_addr, length, rc4_key = struct.unpack("<IQI16s", funcs_start[i*32
        # print(id, hex(start_addr), length, rc4_key.hex())
        funcs['id'] = id
        funcs['start_addr'] = start_addr
        funcs['len'] = length
        funcs['rc4_key'] = rc4_key
        func_list.append(funcs)
    return trap_list, func_list

def unpack(fn, out, runtime=False):
    with open(fn, "rb") as f:
        data = f.read()
    binary = lief.parse(data)
    elf_header = binary.header

    if elf_header.numberof_segments == 2:
        loader_seg = binary.segments[0]
        payload_seg = binary.segments[1]
        payload_offset, payload_size = payload_seg.file_offset, payload_seg.physica
        key_offset = elf_header.program_header_offset + elf_header.program_header_s
        loader_offset, loader_size = key_offset+16, loader_seg.physical_size-key_of
        key = bytearray(data[key_offset:key_offset+16])
        print(key.hex(" "))
        payload = data[payload_offset:payload_offset+payload_size]
        loader = data[loader_offset:loader_offset+loader_size]
        for i, c in enumerate(loader):
            key[i%len(key)] ^= c
        print(key.hex(" "), hex(loader_size), hex(payload_size))
```

```
        rc4 = ARC4.new(key)
        final = rc4.decrypt(payload)
        with open(out, "wb") as f:
            f.write(final)
        if runtime:
            match = re.search(rt_info_pattern, loader, re.DOTALL)
            if match:
                rt_info_offset = match.start()
                trap_list, func_list = rt_info_parser(loader[rt_info_offset:])
                with open(out, "rb") as f:
                    newdata = f.read()
                bin = lief.parse(newdata)
                elf_header = bin.header
                newdata = bytearray(newdata)
                if elf_header.file_type.value == 3:
                    base = 0x800000000
                elif elf_header.file_type.value == 2:
                    base = 0
                for i in func_list:
                    offset = i['start_addr'] - base
                    rc4 = ARC4.new(i['rc4_key'])
                    newdata[offset:offset+i['len']] = rc4.decrypt(newdata[offset:of
                for i in trap_list:
                    offset = i['addr'] - base
                    newdata[offset] = i['value']
                with open(out+".fix", "wb") as f:
                    f.write(newdata)
                print("fixed")

unpack(r"bin.elf","bin.elf.unpack", True)
```

Cases

Currently, Kiteshield's evasion capabilities are highly effective, with most mainstream antivirus software unable to unpack it. The only engine that identifies it provides a generic verdict of "Virus.Generic".

MD5	FIRST SEEN	DETECTION	FAMILY
2c80808b38140f857dc8b2b106764dd8	2023-12-19	1/67	amdc6766
f5623e4753f4742d388276eaee72dea6	2024-05-18	1/67	winnti
4afedf6fbf4ba95bbecc865d45479eaf	2024-05-23	0/67	gafgyt

Below is the comparison of detection rates before and after unpacking Kiteshield-packed ELF files

0x1: APT Level: Winnti

`f5623e4753f4742d388276eaee72dea6` after unpacking has an MD5 hash of `951fe6ce076aab5ca94da020a14a8e1c` and a detection rate of **18/67**. Most antivirus software correctly identifies it as Winnti's userland rootkit.

0x2: Cybercrime Group Level: DarkMosquito (amdc6766)

`2c80808b38140f857dc8b2b106764dd8` after unpacking has an MD5 hash of `a42249e86867526c09d78c79ae26191d` and a detection rate of **0/67**. It belongs to the **cybercrime group amdc6766**, exposed by Sangfor, and functions as a dropper, originating from `s.jpg`.

The fact that it still has zero detections after unpacking was somewhat unexpected for us. It seems that the amdc6766 group has not yet entered the radar of mainstream antivirus software. Here, we directly quote Sangfor's report conclusion.

> *amdc6766 cybercrime grouphas long used methods such as fake pages, supply chain poisoning, and public web vulnerabilities to target software commonly used by IT personnel, such as Navicat, Xshell, LNMP, AMH, OneinStack, and BT. After selecting high-value targets, they implant dynamic link libraries, rootkits, malicious crond services, and other persistent means to control the host for long-term and launch various cybercrime activities.*

0x3: Script Kiddie Level: Gafgyt

`4afedf6fbf4ba95bbecc865d45479eaf` after unpacking has an MD5 hash of `5c9887c51a0f633e3d2af54f788da525` and a detection rate of 23/66. It is a typical Gafgyt botnet.

Conclusion

In recent years, an increasing number of cybercrime actors have turned their attention to the Linux platform, resulting in a surge of Linux malwares. It is evident that Linux malwares are currently experiencing rapid growth. From a security analysis perspective, the countermeasures employed by Linux viruses are relatively "immature" compared to the sophisticated techniques seen in Windows malware.

However, with increased investment from both attackers and defenders, it is foreseeable that Linux malware will evolve to incorporate a variety of advanced anti-analysis techniques and diverse packing methods, similar to those found on the Windows platform. Kiteshield is just the beginning of this trend.Stay Vigilant, Stay Safe.

IOC

MD5

```
Winnti
f5623e4753f4742d388276eaee72dea6
951fe6ce076aab5ca94da020a14a8e1c


Amdc6766
4d79e1a1027e7713180102014fcfb3bf
```

```
2c80808b38140f857dc8b2b106764dd8
a42249e86867526c09d78c79ae26191d

909c015d5602513a770508fa0b87bc6f
57f7ffaa0333245f74e4ab68d708e14e
5ea33d0655cb5797183746c6a46df2e9
7671585e770cf0c856b79855e6bdca2a


Gafgyt
4afedf6fbf4ba95bbecc865d45479eaf
5c9887c51a0f633e3d2af54f788da525
```

Reference

https://www.antiy.com/response/DarkMozzie.html

https://www.freebuf.com/articles/network/401262.html

https://www.freebuf.com/articles/network/401262.html

What do you think?

2 Responses

 Upvote

 Funny

 Love

 Surprised

 Angry

 Sad

0 Comments

Login ▾

G

Start the discussion…

LOG IN WITH

OR SIGN UP WITH DISQUS

Name

Share

Best Newest Oldest

RSS