

Packer

# 警惕：Kiteshield Packer正在被Linux黑灰产滥用



Alex.Turing, Wang Hao

2024年5月28日 • 10 min read



## 背景

[Kiteshield Packer介绍](#)

[初识Kiteshield](#)

[Loader中的奇技淫巧](#)

i. [0x1: 反调试](#)

i. [0x2: 字符串混淆](#)

[检测&脱壳](#)

i. [yara规则](#)

i. [脱壳脚本](#)

[被滥用的案例](#)

[0x1: APT级别: winnti](#)

[总结](#)[IOC](#)[MD5](#)[Reference](#)

# 背景

近一个月XLab的 大网威胁感知系统 捕获了一批VT低检测且很相似的可疑ELF文件。我们在满心期待中开始了逆向分析，期间经历了反调试，字串混淆，XOR加密，RC4加密等等一系列对抗手段。坦白的说，在这个过程中我们是非常开心的，因为对抗越多，越能说明样本的“不同凡响”，或许我们抓到了一条大鱼。

然而，结果却让人失望。这批样本之所以检测率低，是因为它们都使用了一个名为 Kiteshield 的壳。最终，我们发现这些样本都是已知的威胁，分别隶属于**APT** 组织**Winnti**、**黑产团伙暗蚊**，以及某不知名的脚本小子。

尽管我们未能从这批样本中发现新的威胁，但低检测率本身也是一种重要的发现。显然，不同级别的黑灰产组织都开始使用Kiteshield来实现免杀，而目前安全厂商对这种壳还缺乏足够的认知。随着一年一度的大型网络演练临近，我们不排除攻击方使用Kiteshield的可能性。因此，我们认为有必要编写本文，向社区分享我们的发现，以促进杀软引擎对Kiteshield壳的处理能力。

## Kiteshield Packer介绍

简单来说，Kiteshield是一个针对x86-64 ELF二进制文件的加壳器，它使用多层加密来包装ELF二进制文件，并向它们注入加载器代码，该代码完全在用户空间中解密、映射和执行打包的二进制文件。基于ptrace的运行时引擎确保在任何给定时间都只解密当前调用堆栈中的函数，并额外实现了各种反调试技术，以使打包的二进制文件尽可能难以进行反向工程，同时支持单线程和多线程二进制文件。

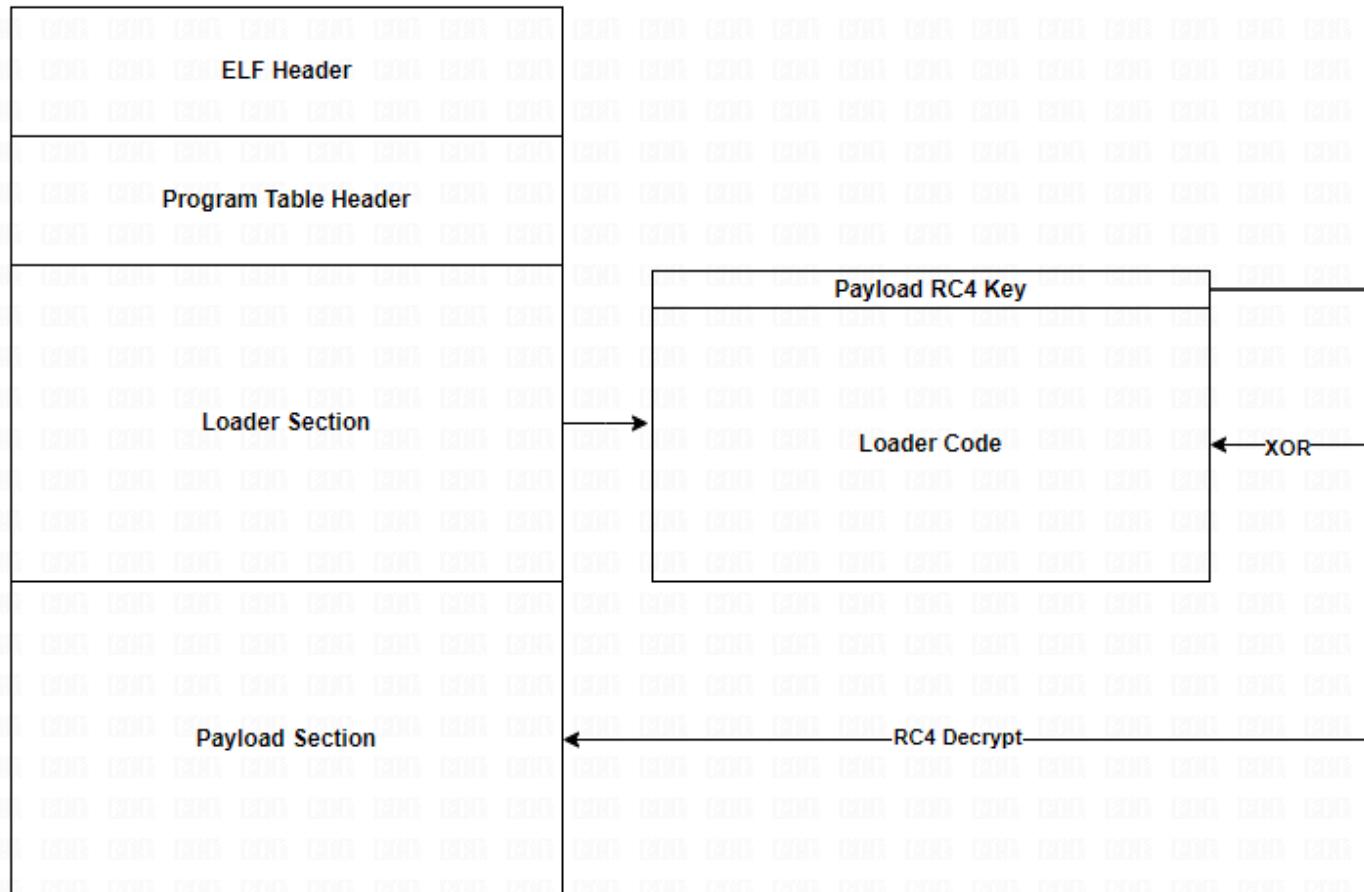
下文将重点说明使用Kiteshield加壳后二进制文件的特征，以及如何脱壳；至于Kiteshield其它的技术细节，感兴趣的读者可以直接参阅其在github的[源代码](#)。

## 初识Kiteshield

Kiteshield加壳的ELF文件只有2个segment，第一个segment为Loader section，第二个为Payload section。

Choose segment to jump												
Name	Start	End	R	W	X	D	L	Align	Base	Type	Class	AD
LOAD	00000000000200000	00000000000201F58	R	W	X	.	L	byte	0001	public	CODE	64
LOAD	00000000000A01F58	00000000000B06C1C	R	W	.	.	L	byte	0002	public	DATA	64
Line 1 of 2												
OK			Cancel			Search			Help			

它们在ELF中的布局如下所示，Loader section使用RC4解密Payload。



初始的rc4\_key位于Program table header结尾与Entry point之间，长度为16字节。

rc4\_key是本身随机生成，不同的样本key不同，以

909c015d5602513a770508fa0b87bc6f 为例，它的的初始rc4\_key为 85 7F 6B

A4 DD 39 5A A1 3E A7 A3 A811 77 E0 8E

初始rc4\_key不能被直接使用，首先它要和Loader的代码进行异或，通过这种方法保证Loader的代码没有被修改，否则无法得到正确的RC4秘钥；然后才是RC4解密，最后将解密后的payload映射到内存中，并跳转到新的入口。

IDA打开KiteShield加壳的文件，入口的代码有非常明显的Pattern。

其中跳转到新的入口的代码 jump to payload section 正是来源于以下源码 loader\entry.S的汇编片段，它是固定不变的，可以当成Kiteshield加壳文件的一个特征。

```
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
```

## Loader中的奇技淫巧

### 0x1: 反调试

Kiteshield使用了4种反调试方法：

1. antidebug\_proc\_check\_traced: 通过检查 `/proc/<pid>/status` 中是否包含 `TracerPid` 字段
2. antidebug\_prctl\_set\_nondumpable: 设置进程的 `dumpable` flag为0, 使 `ptrace`无法附加或转储
3. antidebug\_rlimit\_set\_zero\_core: 设置环境变量 `LD_PRELOAD`, `LD_AUDIT`, `LD_DEBUG` 为空
4. antidebug\_rlimit\_set\_zero\_core: 通过 `setrlimit` 将 `RLIMIT_CORE` (最大核心转储大小)设置为0

## 0x2: 字符串混淆

Kiteshield使用 `DE0BF_STR` 实现单字节异或加密, key硬编码为0x83。

```
#define DE0BF_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; })
```

以下是Loader中被加密的字符串, 主要用于反调试。

```
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'
}
```

从逆向的角度来说，在IDA中可以频繁看到类似的代码片段，用于解密字串。

根据DEOBF\_STR的逻辑，很容易就能编写解密脚本，上图中0x2019E0处的字串解密后为 `/proc/%d/status`。

```
enctext=idc.get_bytes(0x00000000002019E0,15)

plaintext=bytearray()
for idx,value in enumerate(enctext):
    plaintext.append(value ^ (0x83+idx))
print(plaintext)
```

## 检测&脱壳

根据上文提及的入口以及加密字串特征，我们实现了以下yara规则，python脚本分别用于检测，脱壳。

## yara规则

```
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
}
```

# 脱壳脚本

```
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.physical_offset
        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 = 0x8000000000
        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)
```

# 被滥用的案例

目前Kiteshield的免杀效果非常好，主流的杀软几乎都不能脱壳，唯一识别的引擎给出的verdict也是通用型的“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

脱壳前后的检测率对比如下所示：

## 0x1: APT级别：winnti

f5623e4753f4742d388276eaee72dea6 脱壳后MD5为  
951fe6ce076aab5ca94da020a14a8e1c，检测率为18/67，大部分杀软都能正确识别它，正是winnti的[userland rootkit](#)。

## 0x2: 黑产团伙级别：暗蚁 (amdc6766)

2c80808b38140f857dc8b2b106764dd8 脱壳后MD5为  
a42249e86867526c09d78c79ae26191d，检测率为0/67，它隶属于深信服曝光的黑产团伙amdc6766，本身的功能是一个Dropper，来源于s.jpg。。

脱壳后仍然是0检测有点出乎我们的意料，amdc6766组织似乎还没有进入主流杀软的视野，此处我们直接引用[深信服报告的结论](#)。

amdc6766黑产组织长期利用仿冒页面、供应链投毒、公开web漏洞等攻击方式，针对运维人员常用软件Navicat、Xshell、LNMP、AMH、OneinStack、宝塔等开展定向攻击活动，选择出高价值目标后，植入动态链接库、Rootkit、恶意crond服务等持久化手段长期控制主机，伺机发起各类黑产攻击活动。

## 0x3: 脚本小子级别：Gafgyt

4afedf6fbf4ba95bbecc865d45479eaf 脱壳后MD5为

5c9887c51a0f633e3d2af54f788da525，检测率为23/66，是典型的gafgyt僵尸网络。

# 总结

近年来，越来越多的黑灰产组织将目光投向了Linux平台，导致Linux病毒如雨后春笋般不断涌现。显而易见，Linux病毒正处于高速增长的阶段。从安全分析的角度来看，相较于Windows系统上的恶意软件，Linux下的病毒所使用的对抗技术显得相对“稚嫩”。然而，随着双方投入的增加，可以预见，Linux平台上的恶意软件也会像Windows平台一样，充斥着各种奇技淫巧的对抗手段和多样化的壳技术，而Kiteshield，只是一个开始。

# IOC

## MD5

Winnti

f5623e4753f4742d388276eaee72dea6  
951fe6ce076aab5ca94da020a14a8e1c

Amdc6766

4d79e1a1027e7713180102014fcfb3bf

2c80808b38140f857dc8b2b106764dd8  
a42249e86867526c09d78c79ae26191d

909c015d5602513a770508fa0b87bc6f  
57f7ffaa0333245f74e4ab68d708e14e  
5ea33d0655cb5797183746c6a46df2e9  
7671585e770cf0c856b79855e6bdca2a

Gafgyt

# Reference

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

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

## What do you think?

1 Response



Upvote



Funny



Love



Surprised



Angry



Sad

0 Comments

1 Login ▾

G

Start the discussion...

LOG IN WITH

OR SIGN UP WITH DISQUS

Name



Share

Best Newest Oldest

