Cloud Snooper attack bypasses firewall security measures

An investigation of an attack in progress on a cloud computing server reveals an unusual and innovative way for malware to communicate through firewalls

Sergei Shevchenko
Contents

Rootkit in the Cloud .......................................................................................................................... 3
Anomalous traffic raises alerts ......................................................................................................... 3
Dismantling the Cloud Snooper tools ............................................................................................... 3
The Cloud Snooper communications handler .................................................................................. 5
How Cloud Snooper communicates through the firewall ............................................................... 5
  Practical examples of communication with the Snooper ............................................................... 6
Explaining the attack: Wolves in sheep’s clothing .......................................................................... 9
  Technical analysis of Cloud Snooper network operations ............................................................ 11
Sample #3 .......................................................................................................................................... 13
Logging ............................................................................................................................................... 15
  Some additional details on unused/broken 'rootkit' functionality ............................................... 16
Building a Client ............................................................................................................................... 17
Another variant of the rootkit ............................................................................................................. 18
  Sample #4 ......................................................................................................................................... 18
  Sample #5 ......................................................................................................................................... 19
  Sample #6 ......................................................................................................................................... 19
  Sample #7 ......................................................................................................................................... 20
  Sample #8 ......................................................................................................................................... 21
  Sample #9 ......................................................................................................................................... 22
Windows Malware ............................................................................................................................. 23
  Sample #10 ........................................................................................................................................ 23
  Sample #11 - encrypted payload ..................................................................................................... 23
Conclusion ........................................................................................................................................... 28
IOCs ...................................................................................................................................................... 28
Rootkit in the Cloud

In the course of investigating a malware infection of cloud infrastructure servers hosted in the Amazon Web Services (AWS) cloud, SophosLabs discovered a sophisticated attack that employed a unique combination of techniques to evade detection and that permits the malware to communicate freely with its command and control (C2) servers through a firewall that should, under normal circumstances, prevent precisely that kind of communication from reaching the infected server.

Though we discovered the technique in use on AWS, the problem is not an AWS problem per se. It represents a method of piggybacking C2 traffic on a legitimate traffic, such as normal web traffic, in a way that can bypass many, if not most, firewalls.

The complexity of the attack and the use of a bespoke APT (Advanced Persistent Threat) toolset gives us reason to believe that the malware and its operators were an advanced threat actor, possibly nation-state sponsored.

The compromised systems were running both Linux and Windows EC2 instances.

Anomalous traffic raises alerts

As often happens with incidents like this, our investigation started when someone noticed an anomaly. While the AWS Security Groups (SGs) were properly tuned, set up to only allow inbound HTTP or HTTPS traffic, the compromised Linux system was still listening for inbound connections on ports 2080/TCP and 2053/TCP.

An analysis of this system revealed the presence of a kernel-level rootkit that granted the malware's operators the ability to remotely control the server through the AWS SGs. But this rootkit's capabilities are not limited to doing this in the Amazon cloud. It could also be used to communicate with, and remotely control, malware on any server behind any boundary firewall – even an on-premises server.

By unwinding other elements of this attack, we further identified other Linux hosts infected with the same or a similar rootkit.

Finally, we identified a compromised Windows system with a backdoor that communicated with a similar C2 as other compromised Linux hosts, using a very similar configuration format. The backdoor is apparently based on source code of the infamous Gh0st RAT malware.

At this point in the investigation, we still have some open questions. For example, it is still unclear how the attackers managed to compromise the client's system in the first place. One of the working theories is that the attackers broke into a server through SSH protected with password authentication.

Dismantling the Cloud Snooper tools

As you will see from the description below, some samples we collected are directly related to each other, while others belong to a completely different malware family. Nevertheless, all these samples were collected from the same infrastructure, and thus, we consider them part of the same toolset.

Even though some questions remain, we believe it’s important to share all the evidence we have collected with the security community, network administrators, and researchers to raise awareness of this attack.
The description starts with the Linux malware, then progresses into its Windows counterpart that is apparently based on Gh0st RAT.

Overall, we discovered and studied 10 samples in the course of the investigation, which can be broken down as:

<table>
<thead>
<tr>
<th>#</th>
<th>MD5</th>
<th>Name</th>
<th>File size</th>
<th>Platform</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>a3f1e4b337ba1ed35cac3fab75cec369</td>
<td>snd_floppy</td>
<td>738,368</td>
<td>ELF64, x86-64</td>
</tr>
<tr>
<td>2</td>
<td>6a1d21d3fd074520cb6a1fda76d163da</td>
<td>snd_floppy</td>
<td>738,368</td>
<td>ELF64, x86-64</td>
</tr>
<tr>
<td>3</td>
<td>c7a3f6b3c231ad3b683f00edd0e26e4</td>
<td>snoopy</td>
<td>305,309</td>
<td>ELF64, x86-64</td>
</tr>
<tr>
<td>4</td>
<td>9cd93bb2a12cf4ef49ee1ba5bb0e4a95</td>
<td>snd_floppy</td>
<td>544,832</td>
<td>ELF64, x86-64</td>
</tr>
<tr>
<td>5</td>
<td>15e96f0ee3abc9d5d2395c99aabc3b92</td>
<td>vsftpd</td>
<td>60,456</td>
<td>ELF64, x86-64</td>
</tr>
<tr>
<td>6</td>
<td>2b7d54251068a668c4fe8f988bfc3ab5</td>
<td>ips</td>
<td>35,580</td>
<td>ELF32, x86</td>
</tr>
</tbody>
</table>

Linux Malware, Group 2 – Gh0st RAT

<table>
<thead>
<tr>
<th>#</th>
<th>MD5</th>
<th>Name</th>
<th>File size</th>
<th>Platform</th>
</tr>
</thead>
<tbody>
<tr>
<td>7</td>
<td>ecac141c99e8ce83389203b862b24fd</td>
<td>snort</td>
<td>64,412</td>
<td>ELF32, x86</td>
</tr>
<tr>
<td>8</td>
<td>67c8235ac0861c8622ac2ddb1f5c4a18</td>
<td>javad</td>
<td>64,412</td>
<td>ELF32, x86</td>
</tr>
<tr>
<td>9</td>
<td>850bf958f07e6c33a496b39be18752f3</td>
<td>nood.bin</td>
<td>66,000</td>
<td>ELF32, x86</td>
</tr>
</tbody>
</table>

Windows Malware – Gh0st RAT

<table>
<thead>
<tr>
<th>#</th>
<th>MD5</th>
<th>Name</th>
<th>File size</th>
<th>Platform</th>
</tr>
</thead>
<tbody>
<tr>
<td>10</td>
<td>a59c83285679296758bf8589277abde7</td>
<td>NSIPProvider.dll</td>
<td>219,648</td>
<td>PE32, x86</td>
</tr>
<tr>
<td>11</td>
<td>76380f0a8fb56d3bb3c329f193883edf</td>
<td>NSIPProvider.dll.crt</td>
<td>516,097</td>
<td>[encrypted]</td>
</tr>
</tbody>
</table>
The Cloud Snooper communications handler

The central piece of the attack is a file named `snd_floppy` – a kernel module that sets up a network packet filter, using a Netfilter hook (`NF_INET_LOCAL_IN` and `NF_INET_LOCAL_OUT`).

This component was instrumental in giving the malware's operators the ability to communicate with the malware, despite the firewall protecting the AWS EC2 servers.

The two nearly identical samples of `snd_floppy` (file `snd_floppy.ko`) recovered from two different compromised systems are:

**Sample #1**

<table>
<thead>
<tr>
<th>MD5</th>
<th>a3f1e4b337ba1ed35cac3fab75ce369</th>
</tr>
</thead>
<tbody>
<tr>
<td>SHA1</td>
<td>bdd3930938336cc0b1d979d6d40ab0402a4e8c6d</td>
</tr>
<tr>
<td>SHA-256</td>
<td>959796a5b19d61286246ec27e3aef9b8ecc9ea05937da3aa1e98db07df321873</td>
</tr>
<tr>
<td>File size</td>
<td>738,368 bytes</td>
</tr>
<tr>
<td>Internal name</td>
<td><code>snd_floppy</code></td>
</tr>
</tbody>
</table>

**Sample #2** (almost identical to #1):

<table>
<thead>
<tr>
<th>MD5</th>
<th>6a1d21d3fd074520cb6a1fda76d163da</th>
</tr>
</thead>
<tbody>
<tr>
<td>SHA1</td>
<td>d253788241cb52a8c41ff625d423851ba4768545</td>
</tr>
<tr>
<td>SHA-256</td>
<td>a7e288462fbd89758a2783908537e851dfcd841f213266b19c0dbad8827b8682</td>
</tr>
<tr>
<td>File size</td>
<td>738,368 bytes</td>
</tr>
<tr>
<td>Internal name</td>
<td><code>snd_floppy</code></td>
</tr>
</tbody>
</table>

How Cloud Snooper communicates through the firewall

The `NF_INET_LOCAL_IN` is a type of hook that is triggered before the packet reaches the destination port.

The installed hook handler inspects the socket buffer of every IPv4 packet, looking for a command concealed within a header – the command being the `source port number` of the packet originating from the attacker's machine.
An analysis of the malware samples reveal that it is looking for packets that use any of the following source ports. Each unique source port represents a command, such as: 1010, 2020, 6060, 7070, 8080, or 9999.

Firewalls typically prevent machines behind the firewall from receiving traffic sent to arbitrary destination ports, but they don't pay attention to the source ports, because source ports are normally ephemeral, and not relevant to the server or the services it is hosting.

In a typical cloud instance, the server may be set up to receive traffic from any IP address on port 80/TCP (for HTTP) and on 443/TCP (for HTTPS), so the firewall will let any traffic to those ports through to the server. So long as the traffic coming in to one of these standard ports fits the pattern the communications handler is looking for, it will execute one of its built-in instructions. Anything else will be ignored, and the server will serve web pages as normal to browsers.

Practical examples of communication with the Snooper

For example, if the communications handler receives a TCP SYN packet with an origin port of 6060, the malware will decrypt an embedded file (SHA-256: ec22c6e3537fcc0003bb73dd42f41ae077b2cb3ad9cdab295bca46dc91eac1e1) that has been encrypted with RC4 (the key is 'YaHo0@').

It will then drop that decrypted file as /tmp/snoopy, wait for half a second, and then execute it as a usermode application with the call_usermodehelper() syscall. Immediately after that, it deletes the /tmp/snoopy file, so the snoopy application remains running in memory with no physical file present.

The commands passed to two consecutive call_usermodehelper() syscalls are:

```
/bin/sh -c /tmp/snoopy
rm -rf /tmp/snoopy
```

The commands above will run and then delete the file from the filesystem, but an active snoopy process listening on ports 2053 and 2080 remains:

```
If the command is 9999 as a TCP SYN packet, the /tmp/snoopy process self-terminates (in case killall is supported by OS), by passing the following commands to call_usermodehelper() syscall.

```
/bin/sh -c /tmp/snoopy
rm -rf /tmp/snoopy
killall /tmp/snoopy

NOTE: executing snoopy again while it's already running has no effect; by using a file lock mechanism, snoopy makes sure only one instance is running. If that happens, it will output:

```
[ERROR] there is already a instance.
```
Here is the logic of the NF_INET_LOCAL_IN hook handler, which listens for SYN packets sent to the server, using the various source ports:

```python
if tcp:
    if tcp.src_port == 6060:
        if tcp.flags == SYN:
            drop_payload()  # drops/runs snoopy
            return NF_STOP
    elif tcp.src_port == 7070:
        tcp.dst_port = 2080
        adjust_tcp_checksum()
        return NF_STOP
    elif tcp.src_port == 9999:
        if tcp.flags == SYN:
            kill_payload()  # kills snoopy process
            return NF_STOP
    elif tcp.src_port == 2020:
        return NF_STOP
    elif tcp.src_port == 1010:
        tcp.dst_port = 22
        adjust_tcp_checksum()
        return NF_STOP
    else:
        return NF_ACCEPT
elif udp:
    if udp.src_port == 8080:
        udp.dst_port = 2053
        adjust_udp_checksum()
        return NF_STOP
    else:
        return NF_ACCEPT
```

And here is the logic of the NF_INET_LOCAL_OUT hook handler:

```python
if tcp:
    if tcp.dst_port == 7070:
        tcp.src_port = 443  # or, 80 in another variant
```
```python
adjust_udp_checksum()
return NF_STOP
if tcp.dst_port == 2020:
    return NF_STOP
if tcp.dst_port == 1010:
    tcp.src_port = 443  # or, 80 in another variant
    adjust_udp_checksum()
    return NF_STOP
else:
    return NF_ACCEPT
elif upd:
    if udp.dst_port == 8080:
        udp.src_port = 53
    return NF_STOP
else:
    return NF_ACCEPT
```
Explaining the attack: Wolves in sheep's clothing

In the illustration above, our castle represents the targeted server infrastructure; In the case of the incident we investigated, the server was hosted by Amazon Web Services (AWS). At its perimeter, the AWS Security Groups - a set of firewall rules that provide security at the protocol and port access level – limit the inbound network traffic.

For example, you might typically set up an AWS Security Group that only allows web traffic – that is, TCP packets that arrive at ports 80 or 443 – to reach your server. Network traffic with any other destination port never makes it past the SGs.

The infection involves a rootkit that inspects network traffic, and a backdoor that the attackers leverage the rootkit to send commands to, and receive data from, the backdoor.
In order to bypass the AWS Security Groups, depicted here as guards, the attackers communicate with the rootkit by sending innocent-looking requests (depicted in the illustration as a wolf in sheep's clothing) to the web server on the normal web server ports. A listener that inspects inbound traffic before it reaches the web server intercepts the specially-crafted requests, and sends instructions to the malware based on characteristics of those requests.
The listener sends a "reconstructed" C2 command to the backdoor Trojan installed by the rootkit. Depending on the commands included into C2 traffic, the attacker may use the backdoor to steal sensitive data from the target.

The collected data is then delivered back with the C2 traffic. Only this time, the rootkit has to masquerade it again in order to bypass the guards: the wolf dresses itself in sheep's clothing once again. Once outside, the C2 traffic delivers the collected data back to the attackers.

During an entire operation, the normal web traffic, depicted as sheep, keeps flowing to and from the web server through the allowed gate. Visually, the C2 traffic stays largely indistinguishable from the legitimate web traffic.

Technical analysis of Cloud Snooper network operations
To trigger the payload (snoopy) activation, an attacker would send the following packet:
Next, the `snoopy` module would be accessed by the C2, using source port 7070 for TCP-based or 8080 for UDP-based control:

On the way back, the `NF_INET_LOCAL_OUT` hook handler rebuilds the packet again to make sure its source port is restored back to the original port the incoming packet was destined for. This way, the C2 traffic transparently flows through the port(s) allowed by the firewall:

No other Netfilter hooks within the chain, such as `iptables` INPUT/OUTPUT rules, will process the packet if the hook returns `NF_STOP`. This appears to be the purpose of the TCP command 2020: to bypass other Netfilter hooks.

In instances where the Netfilter receives inbound traffic with a source port of 1010/TCP, it directs the contents to the Secure Shell (SSH) port, 22/TCP. For outbound traffic, we have seen two variants using either port 80 or port 443. This will allow for an SSH connection to step around a firewall with IP restrictions on traffic to port 22.
Hence, the ultimate purpose of the snd_floppy rootkit is to provide a covert control channel for the snoopy usermode process, running on a compromised host.

Such covert control channels can be established via any port allowed by firewalls or AWS Security Groups, be it 80, 443, 22, or any other port.

From the outside, the compromised system will show an unusually large volume of traffic that comes from the remote ports 6060, 7070, 8080, and 9999.

But what is the snoopy module? What does it do?

Sample #3

<table>
<thead>
<tr>
<th>MD5</th>
<th>c7a3fefb3c231ad3b683f00edd0e26e4</th>
</tr>
</thead>
<tbody>
<tr>
<td>SHA1</td>
<td>8ba26a4082c80cc216cee89b2033678256f226ef</td>
</tr>
<tr>
<td>SHA-256</td>
<td>ec22c6e3537fcc0003bb73dd42f41ae077b2cb3ad9cdab295bca46dc91eac1e1</td>
</tr>
<tr>
<td>File size</td>
<td>305,309 bytes</td>
</tr>
<tr>
<td>Internal name</td>
<td>snoopy, rrtserver</td>
</tr>
</tbody>
</table>

snoopy is a backdoor trojan that can be executed both as a command line tool and as a daemon (though it needs to be launched with the -d flag for that). The backdoor’s internal version is 3.0.1-2.20170303.

It opens HTTP and/or DNS services on a compromised system, and allows tunneling of the traffic, operating both as a reverse SOCKS5 proxy server, and client.

For example, the incoming control traffic can also be relayed to a different server.

When run with -h option, the tool prints the following syntax:

```
Usage: rrtserver [OPTIONS]

OPTIONS:
    -h
    -d
    -s IPv4[:PORT:{udp|tcp}:{dns|http|none}]
```

Where:

- **-h** option will print out the usage above
- **-d** will run the tool as daemon
--s allows to specify a server address to bind the listening socket to, its port number, and what protocol is used for the traffic: either UDP-based DNS or TCP-based HTTP

The binary requires root privilege; when run, it calls geteuid() to get the user ID. If it fails, it prints the line below and quits:

"Administrator privileges are required."

It sets the working directory to /tmp and obtains a lock for the file /tmp/rrtserver-lock. The lock file is used to make sure there is only one version of the tool running.

The incoming HTTP traffic is accepted on port 2080, and DNS traffic on port 2053.

**NOTE:** the port numbers 2080 and 2053 are default ones; the tool can be executed with different port numbers specified as parameters.

The rootkit parses the received DNS/HTTP traffic to extract hidden commands within it - such commands are called "rrootkit messages" and are distinguished by the presence of a magic header or marker.

For example, to find "rrootkit messages" in HTTP traffic, Snoopy parses the HTTP request to see if it starts with "GET /* HTTP/1.1

rndata:" or "HTTP/1.1 200 OK

rndata:".

Next, the "rrootkit messages" would start from a magic header 0xE381B7F5. If this header is found, such data is called msg-data.

The received msg-data is then decrypted with RC4, using the quite specific key 'A YARN-based system for parallel processing of large data sets'.

The tool then initiates several additional components. These components will process the received msg-data.

Depending on a separate magic header within each msg-data, the data will be processed by a different component.

The initiated components are:

- **view-shell** (magic header 0xFC72E392): pty (pseudo terminal) that allows remote shell
  - the HISTFILE variable is cleared, to make sure /bin/sh execution leaves no history
  - the received commands are then executed with /bin/sh

- **view-file** (magic header 0xFC72E393): file manager that accepts three commands:
  - 'get' - read files
  - 'put' - save file
  - any other command - execute file with popen() syscall

- **view-proxy** (magic header 0xFC72E394): proxy server that accepts the following commands:
  - 'exit' or 'quit' - quit proxy server
  - 'socks5' - starts SOCKS5 proxy server, authentication is provided with user/password passed with the '−u' and '−p' parameters
  - 'rcsocks-cmd: socks is closed.' - closes SOCKS proxy

The SOCKS5 server is based on the open-source sSocks proxy implementation:
sSocks is a package which contains: a socks5 server implements RFC 1928 (SOCKS V5) and RFC 1929 (Authentication for SOCKS V5), a reverse socks server and client, a netcat like tool, which supports socks5 with authentication and a socks5 relay (run a server and send to a another socks5 server).

• **view-pipe** (magic header 0xFC72E398): p2p communicator, that receives commands 'pwd', 'exit', 'quit', 'connect'

On receiving the 'connect' command, it accepts the same parameters as the command-line tool (server IP, port, protocol) and starts tunneling commands to another peer.

The pipe appears to be used to establish connections to other peers.

The negotiation protocol to other peers includes a message 'rrootkit-negotiation: hello'.

Once the connection is established, the logged message displays what peers have been connected, and that a new network node is now open:

"view-pipe: left[address, port]->right[address, port]."

"view-pipe: the network node is opened."

• **view-myproto** (magic header 0xFC72E397): a 'ping'/pong'; depending on a flag it receives, it either:
  o receives a message "rrootkit-negotiation: hello", then responds back "rrootkit-negotiation: ok, go on"
  o checks if the received message was "rrootkit-negotiation: ok, go on"

• **loop-notifier** – creates a pipe, a data channel for inter-process communication (IPC)

The backdoor allows control via IPC pipe as a backup control channel.

Logging

**snoopy** stores many debug messages in clear text.

However, with the internal level of logging set to 0 (none), no debug messages are ever printed. Hence, these debug messages are only used in the testing phase of the malware.

Some of the debug messages are in Chinese:

• 远程路径太长! - The remote path is too long!
• 远程文件不存在! - The remote file does not exist!
• 远程内存空间分配失败! - Remote memory space allocation failed!
• 远程路径不存在! - The remote path does not exist!
• 远程文件已存在! - The remote file already exists!
• 连接失败! - Connection failed!
• 连接成功! - Connection succeeded!
• 参数错误! - Parameter error!

Some messages reveal poor English grammar:
• view don't found
• view-shell: data do not belong to SHELL

If the rootkit is patched so that it always logs debug messages, it will happily print them on screen:

Some additional details on unused/broken 'rootkit' functionality

The tool has an unused (never called) function kernel_load() to drop and load kernel module from its own file. If the module /proc/sys/rrootkit is missing, it drops /tmp/rrtkernel.ko and loads it with:

```
insmod /tmp/rrtkernel.ko 2>/dev/null
```

Next, /tmp/rrtkernel.ko is unlinked, so it's used, temporarily, to drop and then load a kernel module. The function kernel_load() is never called though - there are no references leading to it.

Apart from that, /tmp/rrtkernel.ko is written by reading its own binary file, starting from the file offset of nearly 2GB:

```
readlink("/proc/self/exe", pathname, 0x200uLL);
myfd = open(pathname, 0);
lseek(myfd, g_self_size, SEEK_SET); // file offset is set to offset bytes
```

**where g_self_size is set to 0x72C9E41D:**

```
g_prev_self_size_mark dd 0EF71B69Ah // unused
```

```
g_self_size dd 72C9E41Dh // <-- nearly 2GB
```

```
g_post_self_size_mark dd 33FF0055h // unused
```

This code will not work - even if this function was called (it wasn't), it would have failed.
It would appear the author was tossing around the idea of dropping and loading a kernel module from its own file. There are two unused variables 'prev self size mark' and 'post self size mark' which might indicate some experiments the author has attempted.

The original intention is not clear. However, the presence of markers 'prev self size mark' and 'post self size mark' around the file offset variable is intriguing.

The markers could be used to find the exact location of the offset in the binary:

```
43 44 45 46 47 48 49 4A 4B 4C 4D 4E 4F
0A 0B 0C 0D 0E 0F 10 11 12 13 14 15 16
```

This way, an external patcher could find and modify the actual offset from where the kernel module could be read, saved, and loaded. For example, if it's set to the end of the snoopy binary, the kernel module can thus be read and loaded from the appended data.

However, this feature wasn't implemented, and should, therefore, be considered experimental.

**Building a Client**

By knowing how the C2 protocol works, it is possible to build a client to talk to snoopy either directly, or via snd_floppy rootkit.

What for?

Firstly, the client can ping a host located in the same network to see if it's infected or not.

Secondly, if a host is infected, the client can disinfect it remotely by instructing snoopy to execute its disinfection routine (see the `rmmod` command below – after serving it, the rootkit stopped responding as it was unloaded).

Last but not least, building such a client is cool.

The following screenshot demonstrates the client in action. The `snd_floppy` rootkit intercepts traffic on port 22, even though it's destined for the SSH daemon (seen as `981/sshd` in the snapshot below). Next, it re-routes such traffic internally to the `snoopy` module.
As long as the rootkit is active, the attackers may attempt to smuggle the control traffic through any port allowed by the firewall (the screenshot demonstrates that using ports 21 and 24 makes no difference – these packets are still re-routed by the rootkit to the backdoor).

### Another variant of the rootkit

During the investigation, a different Linux host was found to be running a different variant of the **snd_floppy** rootkit.

#### Sample #4

<table>
<thead>
<tr>
<th>MD5</th>
<th>9cd93bb2a12cf4ef49ee1ba5bb0e4a95</th>
</tr>
</thead>
<tbody>
<tr>
<td>SHA1</td>
<td>b37fdb9d32e90d4be1e0bd187ef64899038c3785</td>
</tr>
<tr>
<td>SHA-256</td>
<td>620616d76334204516501c477ae46953bfc9ab8c29e096ae4f76f2e732e69845</td>
</tr>
<tr>
<td>File size</td>
<td>544,832 bytes</td>
</tr>
<tr>
<td>Internal name</td>
<td>snd_floppy</td>
</tr>
</tbody>
</table>

This kernel module is very similar to the variant described above. The embedded resource is encrypted with the same RC4 key.

The only difference is in the embedded file itself. It is a different file dropped as `/bin/vsftpd`
Sample #5

<table>
<thead>
<tr>
<th>MD5</th>
<th>15e96f0ee3abc9d5d2395c99abc3b92</th>
</tr>
</thead>
<tbody>
<tr>
<td>SHA1</td>
<td>8a34cb3d4431985dafe7e2f9843552b56b2a6641</td>
</tr>
<tr>
<td>SHA-256</td>
<td>cedb5b81f88afbb5b718c3e66ab25bf945476b8e64b1da0f204d3860a694cce5</td>
</tr>
<tr>
<td>File size</td>
<td>60,456 bytes</td>
</tr>
<tr>
<td>Internal name</td>
<td>vsftpd</td>
</tr>
</tbody>
</table>

/bin/vsftpd in this case is not a well-known FTP server daemon but a backdoor that listens on port 2080.

The communications are encrypted with a custom algorithm based on AES and an additional XOR round. Key initialization is based on hashing the string "replace with your password" and a key received from the server, hashed with SHA-1.

The bot can execute three commands, encoded with integer numbers 1 to 3:

1: download specified file
   internal name: tshd_get_file()

2: upload file and save it under a specified file name
   internal name: tshd_put_file()

3: execute remote shell command with /bin/sh
   internal name: tshd_runshell()

A different version of vsftpd, a backdoor, recovered as a file named ips, was found to be listening on port 10443.

Sample #6

<table>
<thead>
<tr>
<th>MD5</th>
<th>2b7d54251068a668c4fe8f988bfc3ab5</th>
</tr>
</thead>
<tbody>
<tr>
<td>SHA1</td>
<td>61f824bd85630b2bcc48defb7b6cb2a963a744c6</td>
</tr>
<tr>
<td>SHA-256</td>
<td>1d6fe2f90b4e05625167c8601f07eb33d9eee4623e3338ef264cc6961f5175a0</td>
</tr>
<tr>
<td>File size</td>
<td>35,580 bytes</td>
</tr>
<tr>
<td>Internal name</td>
<td>ips</td>
</tr>
</tbody>
</table>
The presence of ips suggests that another 32-bit version of snd_floppy that redirects incoming traffic into port 10443 may exist.

The samples #1-#6 described above represent a combination of a rootkit and a passive backdoor that accepts connections on an open port.

Sample #7
Apart from those samples, we have also recovered a different Linux backdoor, a backdoor that does not open any ports. Instead, it relies on a C2 polling mechanism.

The analysis of this bot functionality reveals it belongs to Gh0st RAT, only it’s a version that has been written for Linux.

It is hard to tell if Gh0st has always existed as a multi-platform RAT, or whether the attackers developed a Linux-based Gh0st after the source code of Gh0st for Windows was leaked online.

At the end of the day, it makes sense to have clients deployed across various platforms, using a unified configuration format and C2 protocol, while having a single server for all those clients.

Still, we will leave the guesswork out of this description, rather focusing on what the recovered samples actually do.

<table>
<thead>
<tr>
<th>MD5</th>
<th>ecac141c99e8cef83389203b862b24fd</th>
</tr>
</thead>
<tbody>
<tr>
<td>SHA1</td>
<td>2f4ee1c39f78ecd5a84233233d02b355022aa50</td>
</tr>
<tr>
<td>SHA-256</td>
<td>c49371cd8dd33f725a780ea179e6281f5cb7f42e84a00836c8fe3350b7b9b2d0</td>
</tr>
<tr>
<td>File size</td>
<td>64,412 bytes</td>
</tr>
<tr>
<td>File Name</td>
<td>/bin/snort</td>
</tr>
</tbody>
</table>

/bin/snort is a backdoor that contacts a remote C2 to fetch and execute commands. Its internal config file is encrypted with RC4, using the password: "r0st#@$":

```
185.86.151.67:443;|1;1;1;1;1;0;0;|--0;|10
```

The '1;1;1;1;1;0;0;' part of the config are the flags that stand for seven days of the week.

The '10-20;' seems to indicate working hours (10 a.m. to 8 p.m.), so current weekday and current hour should match what's in config.

If there is no match, the bot falls asleep for just over seven minutes (423.756 seconds), then checks the time again.

In case of a match, it attempts to reach the C2; if it cannot, it retries again in one minute.
Traffic to the C2 is encrypted with double RC4, where a key is randomly generated based on the current time.

The backdoor has six commands:

- The bot clears environmental variable `HISTFILE` to make sure no history is kept for `/bin/bash` execution; the C2 responds with a string, and the bot sets `TERM` variable to that returned string.

  Next, it receives a command and executes it with `/bin/bash`, with or without a `-c` switch (allows for executing commands as provided within the quotes).

  The output from the executed command is sent back.

- File manipulations:
  - Locate and obtain timestamp for the specified file
  - Rename specified file
  - Recursively delete all files in the specified directory

- More file manipulations:
  - Read the contents of the specified file
  - Recursive search for files
  - Write data into a specified file
  - Create specified directory

- The next two commands manipulate file descriptors with `fcntl()` syscall, and fork child processes.

- Receive data and save it into a local file `/usr/include/sdfwex.h`

It appears that `/usr/include/sdfwex.h` contains a timestamp (year, month, day, hour, minutes) for when the C2 connection should commence.

If the bot cannot open this file, it tries to open `/tmp/.llock`—if that file also cannot be opened, the bot skips the timestamp check, and proceeds with trying to connect to the C2.

Sample #8

A backdoor very similar to `/bin/snort` was recovered as `/usr/bin/javad`, described below:

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>MD5</strong></td>
<td>67c8235ac0861c8622ac2ddb1f5c4a18</td>
</tr>
<tr>
<td><strong>SHA1</strong></td>
<td>6aa0b6bfe059354782febd4fa665dbacd726b488</td>
</tr>
<tr>
<td><strong>SHA-256</strong></td>
<td>a8db92a8f34caa5084a3fdb8a683a1854bff84612dfd25a965bc12a454a38556</td>
</tr>
<tr>
<td><strong>File size</strong></td>
<td>64,412 bytes</td>
</tr>
<tr>
<td><strong>File Name</strong></td>
<td><code>/usr/bin/javad</code></td>
</tr>
</tbody>
</table>
This backdoor is similar to the sample #7 (c49371cd8dd3f725a780ea179e6281f5cb7f42e84a00836c8fe3350b7b9b2d0). It uses a different configuration:

\[
\text{cloud.newsofnp.com:443;|1;1;1;1;1;1;1;|00-24;|1}
\]

An analysis of network activity revealed that a similar domain – ssl.newsofnp.com was also resolved from a Windows host.

Sample #9

A backdoor very similar to /bin/snort and /usr/bin/javad:

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>MD5</td>
<td>850bf958f07e6c33a496b39be18752f3</td>
</tr>
<tr>
<td>SHA1</td>
<td>ea579984897dd585af348ecbfc112044a0346ca1</td>
</tr>
<tr>
<td>SHA-256</td>
<td>dbd926b097e5a2b142b89f9ce94fd076b0c6283f0e38a1c6ce01ab87cf41edd</td>
</tr>
<tr>
<td>File size</td>
<td>66,000 bytes</td>
</tr>
<tr>
<td>File Name</td>
<td>nood.bin</td>
</tr>
</tbody>
</table>

Just like other samples, it decrypts its config file using RC4 key "r0st@#$".

The decrypted config is:

\[
\text{load.CollegeSmooch.com:82;|1;1;1;1;1;1;1;|00-24;|10}
\]

Just like /bin/snort and its Windows counterpart NSIProvider.dll, it also checks if the current day matches the configuration file and falls asleep for exactly 423.756 seconds (just over seven minutes) before it tries again.

For the beacon signal it sends to the C2, it collects basic system configuration into a fingerprint. This info consists of:

- Hostname and IP address
- Platform type, as read from /proc/version, such as 'x86_64'
- Full name of the Linux version, as read from /etc/issue.net and /etc/issue, such as:
  - 'Red Hat Enterprise Linux Server release 6.10 (Santiago)'
  - or
  - 'Ubuntu 16.04.5 LTS'
The communications with the C2 are always encrypted using a bespoke algorithm that relies on a time-based random RC4 key with extra encryption layers.

The backdoor commands received from the C2 and executed by `nood.bin` fully match `/bin/snort` functionality. That is, it provides remote shell and a dedicated remote file manipulation capability, such as an ability to read, write, rename, delete, or recursively search for files.

**Windows Malware**

**Sample #10**

<table>
<thead>
<tr>
<th>MD5</th>
<th>a59c83285679296758bf8589277abde7</th>
</tr>
</thead>
<tbody>
<tr>
<td>SHA1</td>
<td>2ff1ff96fe83c607c8b7a4a279b6bc3103de1d33</td>
</tr>
<tr>
<td>SHA-256</td>
<td>f7d69c21c683e19624169d3bc70d06b2896a9ccf6186301d16f500db520d3b19</td>
</tr>
<tr>
<td>File size</td>
<td>219,648 bytes</td>
</tr>
<tr>
<td>File Name:</td>
<td>C:\ProgramData\NSIProvider\NSIProvider.dll</td>
</tr>
</tbody>
</table>

**Sample #11 - encrypted payload**

<table>
<thead>
<tr>
<th>MD5</th>
<th>76380fea8fb56d3bb3c329f193883edf</th>
</tr>
</thead>
<tbody>
<tr>
<td>SHA1</td>
<td>cd848eb12be9588609ed9e5afad74adfd5c3798a</td>
</tr>
<tr>
<td>SHA-256</td>
<td>2bc8fea05b1e9409c3cf065a30aa450ace911d91fdbb55ad282c2925a9aac766</td>
</tr>
<tr>
<td>File size</td>
<td>516,097 bytes</td>
</tr>
<tr>
<td>File Name:</td>
<td>C:\ProgramData\NSIProvider\NSIProvider.dll.crt</td>
</tr>
</tbody>
</table>

NSIProvider.dll is a malicious Windows service DLL, executed under `svchost.exe`.

The service name is NSIProvider, registered with the description name "Netword Store Interface Provider."

**NOTE:** 'Netword' with 'd'.
The DLL is heavily obfuscated.

Once started as a service, it conveniently spits out debug messages documenting the operation.

Sysinternal's DebugView shows these messages:

<table>
<thead>
<tr>
<th>Offset</th>
<th>Time (ms)</th>
<th>Message</th>
</tr>
</thead>
<tbody>
<tr>
<td>00000000</td>
<td>0.00000000</td>
<td>DLL_PROCESS_ATTACH.</td>
</tr>
<tr>
<td>00000001</td>
<td>0.00489140</td>
<td>Rundll32Entry()</td>
</tr>
<tr>
<td>00000002</td>
<td>0.01733349</td>
<td>ServerLoadPayload()</td>
</tr>
<tr>
<td>00000003</td>
<td>0.01749189</td>
<td>Get Module File Name.</td>
</tr>
<tr>
<td>00000004</td>
<td>0.01753826</td>
<td>Get Payload File Name.</td>
</tr>
<tr>
<td>00000005</td>
<td>0.01757095</td>
<td>Switch to payload directory.</td>
</tr>
<tr>
<td>00000006</td>
<td>0.01768074</td>
<td>Read Payload File.</td>
</tr>
<tr>
<td>00000007</td>
<td>0.01811264</td>
<td>Decrypt Payload Data.</td>
</tr>
<tr>
<td>00000008</td>
<td>0.06122175</td>
<td>Verify Payload Data.</td>
</tr>
<tr>
<td>00000009</td>
<td>0.06732560</td>
<td>ServerExecutePayload()</td>
</tr>
<tr>
<td>00000010</td>
<td>0.06740102</td>
<td>Call Shellcode.</td>
</tr>
</tbody>
</table>

Once loaded, the DLL locates the encrypted payload file and loads it into memory.

The steps are:

- Get current module filename with `GetModuleFileName()` API, i.e. `%PATH%\NSIProvider.dll`
- Concatenate current module filename with '.crt', e.g. `%PATH%\NSIProvider.dll.crt`
- Allocate memory `VirtualAlloc()` and read the entire payload file into memory
- Initialise a permutation table that consists of 256 DWORDs
  Each value of the permutation table is calculated as:

  ```c
  *ptr = ((*ptr >> 1) & 0x54384748 | ~(*ptr >> 1) & 0xABC7B8B7) ^ 0x467F3B97;
  ...;
  PERM_TABLE[*index] = *ptr;
  ```
- Start decryption loop – in this loop, a key value is subtracted from each byte of the encrypted payload; the key value itself is calculated in each iteration based on the previous key value, current index of the decrypted byte, and the permutation table:
ptr = __ptr_index++;

val = PERM_TABLE((*ptr & 0x67612505 | ~*ptr & 0x989EDAFA) ^ (KEY & 0x67612505 | ~KEY & 0x989EDAFA)) & ((~*ptr & 0x67612505 | ~^*ptr & 0x989EDAFA) ^ 0xFFFF00));

KEY = (val & 0x432AA81D | ~val & 0xBCD557E2) ^ ((KEY >> 8) & 0x432AA81D | ~(KEY >> 8) & 0xBCD557E2);

- The decrypted payload reveals a checksum, a number of zero bytes, followed by the initial shellcode itself:

The decrypted payload blob is copied into a newly allocated memory buffer and the initial shellcode (starts from bytes EB 17 58 in the image above) is called.

The initial shellcode will then decrypt the rest of the blob using an XOR key that starts from 0x2B, and then incremented by the index of the decrypted byte, i.e. the XOR key values are: 0x2B, 0x2C, 0x2E, 0x31, etc.

As the rest of the blob is decrypted, the configuration file is decrypted as well, followed by other parts. After the initial shellcode has finished the decryption, the fully decrypted blob will consist of:

- Initial shellcode
- Decrypted config:

```
Microsoft.Windows.BNG|ssl.newsofnp.com:443;|1;1;1;1;1;1;00-24;|1
```

- Zlib-compressed LIBEAY32.dll (77,871 bytes, 167,936 bytes when decompressed)
- Zlib-compressed LIBEAY32.dll (386,876 bytes, 851,968 bytes when decompressed)
- Backdoor, implemented in the form of a second-stage shellcode

Once it’s decoded, the second-stage shellcode is called – this is the backdoor itself.
When it gets control, it dynamically obtains all the APIs it needs by using hard-coded API hashes. To find matching hashes from the API names, the shellcode relies on a slight modification of the ROR-13 algorithm. The only difference is that it checks if the zero byte character is at the end of the loop, and thus has an additional ROR for the terminating zero byte.

All the required DLLs are loaded dynamically.

Next, it will decompress and load two stubs as DLLs. Both DLLs have the internal name LIBEAY32.dll. Both DLLs rely on an older (2004) build of the libeay32.dll. Below are some strings found in the body of these DLLs:

<table>
<thead>
<tr>
<th>String</th>
<th>Version</th>
<th>Date</th>
</tr>
</thead>
<tbody>
<tr>
<td>MD2 part of OpenSSL 0.9.7d 17 Mar 2004</td>
<td></td>
<td></td>
</tr>
<tr>
<td>MD4 part of OpenSSL 0.9.7d 17 Mar 2004</td>
<td></td>
<td></td>
</tr>
<tr>
<td>MD5 part of OpenSSL 0.9.7d 17 Mar 2004</td>
<td></td>
<td></td>
</tr>
<tr>
<td>SHA part of OpenSSL 0.9.7d 17 Mar 2004</td>
<td></td>
<td></td>
</tr>
<tr>
<td>SHA1 part of OpenSSL 0.9.7d 17 Mar 2004</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

The backdoor relies on these DLLs for crypto-functions required to communicate with the C2.

The config format is consistent with the ELF binaries, i.e., the seven '1;' means the bot should be active seven days a week, all hours (00-24), the C2 communicates via HTTPS.

The same config is known to be used by the Gh0st RAT.

Just like /bin/snort described above, the bot also checks if the current day and hour match what's specified in the config.

If there is no match, the bot also falls asleep for just over seven minutes (423.756 seconds), then checks the time again.

The code snippets below demonstrate that the 423,756-millisecond delay specified within /bin/snort executable is identical to its Windows counter-part:

**ELF executable:** /bin/snort

**Windows shellcode:**
On Linux, the number 423,756 is multiplied by 1,000, then passed to `usleep()` syscall that takes an argument in microseconds.

On Windows, the same number is passed to `Sleep()` API, which takes the argument in milliseconds.

In both cases, the achieved delay is identical: 7.062 seconds.
Conclusion

This case is extremely interesting as it demonstrates the true multi-platform nature of a modern attack. A well-financed, competent, determined attacker will unlikely ever to be restricted by the boundaries imposed by different platforms.

Building a unified server infrastructure that serves various agents working on different platforms makes perfect sense for them.

When it comes to prevention against this attack (or similar ones), AWS Security Groups provide a robust boundary firewall for EC2 instances. However, this firewall does not eliminate the need for network administrators to keep all external-facing services fully patched.

The default installation for the SSH server also needs extra steps to harden it against attacks, turning it into a rock-solid communication daemon.

IOCs

Ports open on a local host:
- tcp 2080
- udp 2053
- tcp 10443

Example:
user@host:~$ sudo netstat -peanut | grep ":2080 \|:2053 "
tcp 0 0 0.0.0.0:2080 0.0.0.0:* LISTEN 0 34402 2226/snoopy
udp 0 0 0.0.0.0:2053 0.0.0.0:* 0 34398 2224/snoopy

To check if these ports are open on a remote compromised host with IP 192.168.5.150:
user@host:~$ sudo nmap 192.168.5.150 -p 2080
...  
PORT     STATE    SERVICE
2080/tcp filtered autodesk-nlm

user@host:~$ sudo nmap 192.168.5.150 -p 2053 -sU
...  
PORT     STATE    SERVICE
2053/udp filtered lot105-ds-upd

Inbound connections from the remote ports:
6060, 7070, 8080, 9999, 2020, 1010
Domains:
cloud.newsofnp.com
ssl.newsofnp.com
load.CollegeSmooch.com

IPs:
62.113.255.18
89.33.246.111
185.86.151.67

Filenames:
/tmp/rrtserver-lock
/proc/sys/rrootkit
/tmp/rrtkernel.ko
/usr/bin/snd_floppy

Kernel module:
snd_floppy

Example:
user@host:~$ sudo lsmod | grep "snd_floppy"
snd_floppy 316594 0

Syslog message:
"...insmod: ERROR: could not insert module /usr/bin/snd_floppy: File exists"
"...kernel: snd_floppy: loading out-of-tree module taints kernel."
"...kernel: snd_floppy: module verification failed: signature and/or required key missing — tainting kernel"