The Sulley Framework: Basics

By loneferret
www.kioptrix.com

Entirely coded using Python, "The Sulley Framework" was developed by Tipping-Point's Pedram Amini and Aaron Portmoy. Sulley is a fuzzer packed with interesting capabilities. Such as packet-capturing, crash reporting and VMware automation. It also has the capability to restart the target application in the event of a crash and continue on with the fuzzing process.

For data generation, Sulley uses a block-based fuzzing, the same method used by Dave Aitel's SPIKE. So if you're familiar with SPIKE, you shouldn't have much trouble with this fuzzer. In block based fuzzing, you build up a general skeleton for the protocol or file you are fuzzing.

One great feature this fuzzer has over others is the ability to show the CPU's registers at the moment of the crash. Sulley comes with PyDBG, and once the framework installed it uses this to monitor the process you are fuzzing. Another one is Sulley's ability to monitor the network (using pcapy), capturing every "fuzz" sent to the target. This way you can follow your progress, or review your fuzzing session. Lastly, you can even see the fuzzing progress via an HTML page. The page shows you which variable it's currently fuzzing and its progress.

Sulley Primitives

Strings:
Sulley uses the "*s_strings()*" directive to denote that the data contained is a fuzzable string. So if we wanted to fuzz an email address, we'd declare it like this in our skeleton:

```
s_string("fuzzing@sulley.com")
```

This tells Sulley it's a valid value, so it will fuzz that string and exhaust all reasonable possibilities. Once it's finished, it will revert to its original value and continue fuzzing the rest of the declared values down the file.

Delimiters:
This is just a small string that helps break up larger ones. If we take our email example from above, the delimiter would be the "@" sign. This is how we'd pass this primitive on to Sulley:

```
s_string("fuzzing")
s_delim("@")
s_string("sulley")
s_delim(".")
s_string("com").
```

Like the s_string() primitive, the delimiter is also fuzzed. If we don't need to fuzz a particular delimiter, we can tell Sulley to not fuzz it like so:

```
s_delim( ".", fuzzable=false )
```

Static:
The values passed to a static string, remain unchanged (or un-fuzzed I suppose).

```
s_static("\r\n")
```

So let's examine a complete yet small and simple skeleton file. In this example, we'll look at the FTP protocol.

```
# import all of Sulley's functionality
# We'll call this file ftp_ability.py
from sulley import *

s_initialize("user")
s_static("USER")
s_delim(" ")
s_static("ftp")
s_static("\r\n")

s_initialize("pass")
s_static("PASS")
s_delim(" ")
s_static("ftp")
s_static("\r\n")

s_initialize("stor")
s_static("STOR")
s_delim(" ")
s_string("AAAA")
s_static("\r\n")
```

You can see it's pretty simple. This file basically feeds Sulley with valid starting values and it takes it from there. Fuzzing the fields you denoted as fuzzable. In this case; the username, password and the STOR command. Depending on the target FTP server, you'll add and/or remove commands. Not every FTP server will have "STOR", and not all of them have "LIST" … So this is where you'd supply a valid list of commands.

Now that we've seen our blocks, let's take a look at the session file. This file starts the whole fuzzing process.

```
from sulley import * # import everything from Sulley
from requests import ftp_ability # this is our ftp_ability.py file from requests folder

def receive_ftp_banner(sock):
sock.recv(1024)
sess = sessions.session(session_filename="audits/ability.session")

#Target IP xxx.xxx.xxx.xxx
target = sessions.target("xxx.xxx.xxx.xxx", 21)
target.netmon = pedrpc.client("xxx.xxx.xxx.xxx", 26001)
target.procmon = pedrpc.client("xxx.xxx.xxx.xxx", 26002)
target.procmon_options = { "proc_name" : "Ability Server.exe" }

sess.pre_send = receive_ftp_banner #grab the banner
sess.add_target(target)

sess.connect(s_get("user")) # Notice our commands from the previous file
sess.connect(s_get("user"),s_get("pass"))

# This tells Sulley user must be authenticated to use this command
sess.connect(s_get("pass"),s_get("stor"))
sess.fuzz()
```

Because most FTP servers send a banner, we tell Sulley to wait for it before fuzzing any data. The next thing is the session file which keeps track of our overall session. This allows us to stop and restart our fuzzing where we had previously left off.

After that we define our target with the appropriate IP and port number. In this case port 21. After defining the target, we tell our network sniffer on the same host and listening on 26001. Last we tell our debugger is listening on the same host and that is listening on 26002.
We chain in the authentication commands and tell Sulley to start fuzzing.

To start the fuzzing process, we need to have Sulley installed on 2 machines (I find it less confusing this way and it just works better), the attacking and victim machine.
From the target machine:

```
1: start the process we want to fuzz (this case Ability Server)
2: attach the process monitor to our server.
C:\sulley>python process_monitor.py -c c:\ability.crash -p "Ability Server.exe"
3: attach the network monitor to our network card and have it sniff for specific traffic. The -P
parameter is to store your pcaps file. You must create this folder first. You can use a mapped
drive too.
C:\sulley>python network_monitor.py -d 1 -f "src or dst port 21" -P z:\
```

From our attacking machine:

1: execute the session file from Sulley's root directory
C:\sulley>python ftp_session_ability.py

Once the process has started, you can point your browser to the attacking machine's IP on port 26000 to get a progress report. You need to manually refresh the page, or you can easily modify the code to refresh automatically. Once the application crashes, you'll be able to see the report on the page. Enabling you to see what/where was overwritten in the CPU's registry.

You can get the install executable here
And the PDF here

Here's a link to a video on kioptix.com demonstrating all of this. Using the same files and vulnerable application mentioned above. Thanks to dookie for pointing out a few mistakes of mine while I was setting up Sulley for the first time.

Let's move on and look at fuzzing a TFTP server. The big difference is one uses the TCP protocol and the other UDP.

By default Sulley will connect to TCP ports. We need to specify that we are trying to fuzz UDP. This is specified in our session file.

```
from sulley import * # import everything from Sulley
from requests import tftp
sess = sessions.session(session_filename="audits/tftpserver.session",proto="udp")
#Target IP xxx.xxx.xxx.xxx
target = sessions.target("xxx.xxx.xxx.xxx", <PORT#>)
target.netmon = pedrpc.client("xxx.xxx.xxx.xxx", 26001)
target.procmon = pedrpc.client("xxx.xxx.xxx.xxx", 26002)
target.procmon_options = { "proc_name" : "<PROCESS NAME>" }
sess.add_target(target)
sess.connect(s_get("tftp"))
sess.fuzz()
```

Once you've specified the "proto" parameter, the rest of the session file is pretty much the same as fuzzing any other protocol. Now that you have your session file configured for UDP connections, you'll need a request file. I found this basic file TFTP request file on the Internet here. You can also find a few more examples here: http://sulley.kioptrix.com/request.php

Now that we have our session and request file, there's one more change that needs to be done before we can appreciate all of this. When fuzzing a TCP protocol, you would run the network_monitor script like so:

```
c:\sulley>python network_monitor.py -d X -f "src or dst port XX" -P \\path
```

Well since this is UDP and the traffic is only one way, the pcap string won't capture anything. So you'll need to enter it this way:

```
c:\sulley>python network_monitor.py -d X -f "udp dst port XX" -P \\path
```

As with anything script related, this can be improved.
So know you can pretty much continue on fuzzing the TFTP server in very much the same way as the FTP example above.

Try downloading a known vulnerable TFTP server and watch it fuzz… Here's a nice little list from exploit-db that you can have fun with.

Thanks again, hope you enjoyed this little read.

Special thanks to the Exploit-DB Team @ www.exploit-db.com