Sulley fuzzer learning---2

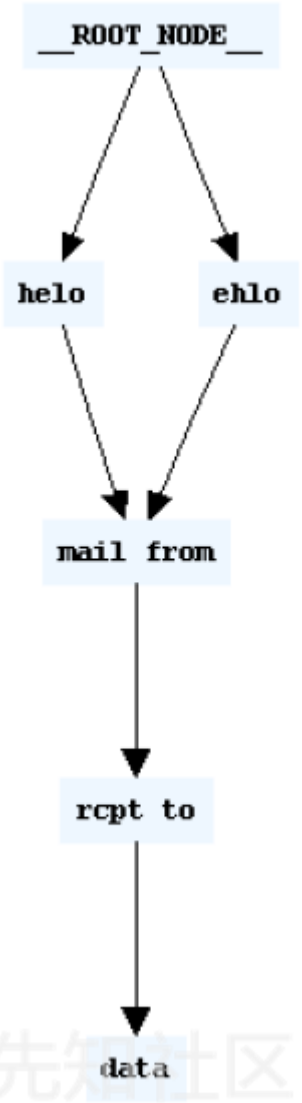
行之 / 2018-08-15 10:30:31 /

前言

这里时Sulley使用手册的第二部分。
手册链接：http://www.fuzzing.org/wp-content/SulleyManual.pdf

Sessions

一旦定义了许多request，就可以在session中将它们绑定在一起了。
Sulley相对于其他fuzz框架的主要优点之一是它能够在协议中"深入"fuzz，这是通过将request链接在一个graph(图)中实现的。在下面的示例中，request序列被绑定在一起




```
from sulley import *
s_initialize("helo")
s_static("helo")
s_initialize("ehlo")
s_static("ehlo")
s_initialize("mail from")
s_static("mail from")
s_initialize("rcpt to")
s_static("rcpt to")
s_initialize("data")
s_static("data")
sess = sessions.session()
sess.connect(s_get("helo"))
sess.connect(s_get("ehlo"))
```

```
sess.connect(s_get("helo"), s_get("mail from"))
sess.connect(s_get("ehlo"), s_get("mail from"))
sess.connect(s_get("mail from"), s_get("rcpt to"))
sess.connect(s_get("rcpt to"), s_get("data"))
fh = open("session_test.udg", "w+")
fh.write(sess.render_graph_udraw())
fh.close()
```

fuzz时，Sulley从根节点开始遍历graph结构，并沿途fuzz每个组件。这个例子以'helo' requet开头。一旦完成，Sulley将开始fuzz “maiL from”
request。它通过在每个测试用例前面加上有效的“helo” request来实现。 接下来，Sulley继续fuzz 'rcpt to’
request。同样，这是通过在每个测试用例前面加上有效的“helo”和“mail from” request来实现的。等到该过程一直到”data”
request完成后,又转回到从”ehlo”开始。通过构建的graph将协议分解为单个请求，并fuzz所有可能路径，使得Sulley的fuzz能力十分强大。举个实际例子，2006年9月针对I
Collaboration Suite的问题。在这种情况下，软件故障是在解析字符'@'和'：'中包含的长字符串时造成的堆栈溢出。
这个案例的有趣之处在于，这个漏洞只暴露在'ehlo'路线而不是'helo'路线上。 如果我们的fuzzer无法走完所有可能的协议路径，那么可能会错过像他这样的问题。

实例化session时，可以指定以下可选关键字参数：

- session_filename :(string，default = None ）（ Filename to serialize persistent data to. ）。 指定文件名来操作fuzzer的执行。
- skip :(整数，默认= 0 ）要跳过的测试用例数。
- sleep_time :(float，default = 1.0 ）在传输测试用例之间sleep的时间。
- log_level :(整数，默认= 2 ）设置日志级别，更高的数字==更多日志消息。
- proto :(string，default =“tcp” ）通信协议。
- timeout：（ float，default = 5.0 ）设置等待send（ ）/ recv（ ）的超时时长。
- restart_interval: (integer, default=0) 在N个测试用例之后重新启动目标，通过设置为0禁用
- crash_threshold :(整数，默认值= 3 ）节点耗尽前允许的最大崩溃次数

Sulley引入的另一个高级功能是能够在协议图结构中定义的每个边缘上注册回调。这允许我们注册在节点传输之间调用的函数，以实现诸如质询响应系统之类的功能。回调方

```
def callback(node, edge, last_recv, sock)
```

'node'是要发送的节点，'edge'是当前fuzz到node'的最后一条路径，'last_recv'包含从最后一个套接字传输返回的数据，'sock'是实时套接字。如果你需要动态填写目标IP地

Targets and Agents

下一步是定义target，将它们与代理链接并将目标添加到会话中。在以下示例中，我们实例化一个在VMWare虚拟机内运行的target，并将其链接到三个代理：

```
target = sessions.target("10.0.0.1", 5168)
target.netmon = pedrpc.client("10.0.0.1", 26001)
target.procmon = pedrpc.client("10.0.0.1", 26002)
target.vmcontrol = pedrpc.client("127.0.0.1", 26003)
target.procmon_options = \
{
"proc_name" : "SpntSvc.exe",
"stop_commands" : ['net stop "trend serverprotect"'],
"start_commands" : ['net start "trend serverprotect"'],
}
sess.add_target(target)
sess.fuzz()
```

实例化的目标绑定在主机10.0.0.1上的TCP端口5168上。
网络监视器代理正在目标系统上运行，默认侦听端口26001.网络监视器将记录所有套接字通信到由测试用例编号标记的各个PCAP文件。
进程监视器代理程序也在目标系统上运行，默认情况下在端口26002上进行侦听。此代理接受指定要附加到的进程名称的其他参数，用于停止目标进程的命令和用于启动目标
最后，VMWare控制代理在本地系统上运行，默认情况下在端口26003上进行侦听。目标将添加到会话中并开始fuzz。
Sulley可以fuzz多个target，每个target都有一组独特的链接代理。 这样分割测试空间（total test space）可以有效节省fuzz时间。

让我们仔细看看每个代理（agnet）的功能。

Agent: Network Monitor (network_monitor.py)

网络监视器代理负责监视网络通信并将其记录到磁盘上的PCAP文件。 代理硬编码绑定到TCP端口26001，并通过PedRPC自定义二进制协议接受来自Sulley
session的连接。在将测试用例发送给target之前，Sulley联系该代理并请求它开始记录网络流量。
一旦测试用例成功传输，Sulley再次联系该代理，请求它将记录的流量刷新到磁盘上的PCAP文件。
PCAP文件以测试用例编号命名，便于检索。此代理不必在与目标软件相同的系统上启动。 但是，它必须能够看到发送和接收的网络流量。
此代理接受以下命令行参数：

```
ERR> USAGE: network_monitor.py
<-d|--device DEVICE #> device to sniff on (see list below)
[-f|--filter PCAP FILTER] BPF filter string
[-p|--log_path PATH] log directory to store pcaps to
```

```
[-l|--log_level LEVEL] log level (default 1), increase for more verbosity
Network Device List:
[0] \Device\NPF_GenericDialupAdapter
[1] {2D938150-427D-445F-93D6-A913B4EA20C0} 192.168.181.1
[2] {9AF9AAEC-C362-4642-9A3F-0768CDA60942} 0.0.0.0
[3] {9ADCDA98-A452-4956-9408-0968ACC1F482} 192.168.81.193
...
```

Agent: Process Monitor (process_monitor.py)

过程监控代理负责检测fuzz target过程中可能发生的故障。 代理硬编码绑定到TCP端口26002并通过PedRPC自定义二进制协议接受来自Sulley session的连接。在成功将每个单独的测试用例传输到target之后，Sulley联系该代理以确定是否触发了故障。如果是这样，关于故障性质的hgih level信息将被传送回Sulley session，以便通过内部Web服务显示（稍后将详细介绍）。

```
ERR> USAGE: process_monitor.py
<-c|--crash_bin FILENAME> filename to serialize crash bin class to
[-p|--proc_name NAME] process name to search for and attach to
[-i|--ignore_pid PID] ignore this PID when searching for the target process
[-l|--log_level LEVEL] log level (default 1), increase for more verbosity
```

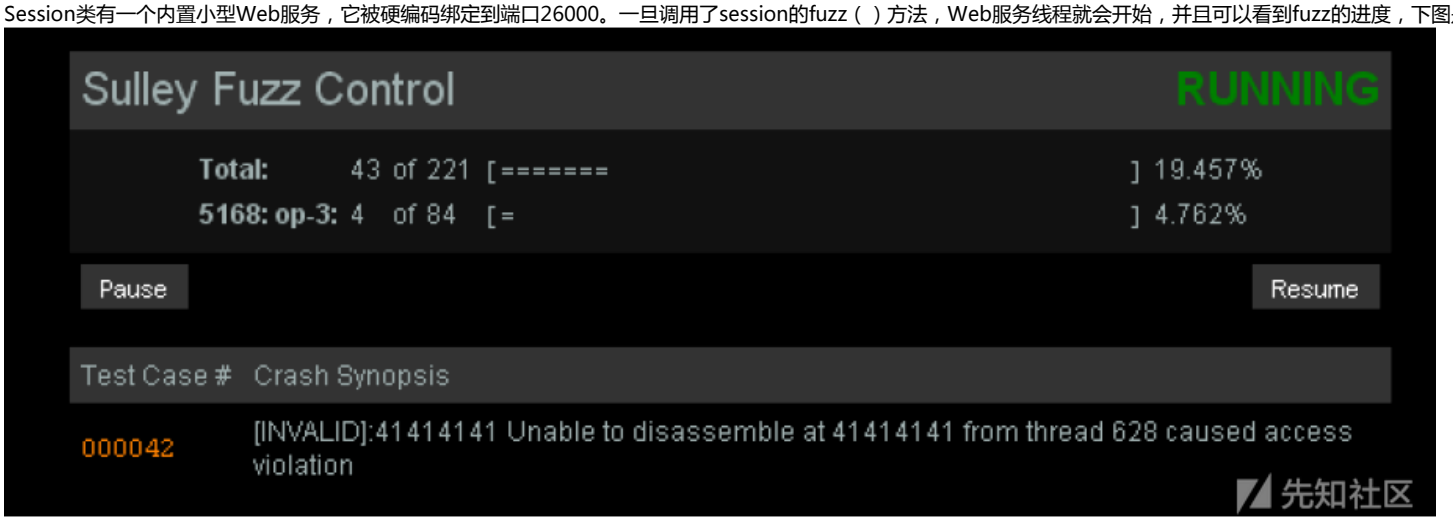
Agent: VMWare Control (vmcontrol.py)

VMWare控制代理硬编码为绑定到TCP端口26003并接受来自Sulley的连接通过PedRPC自定义二进制协议进行会话。
此代理公开用于与虚拟机映像交互的API，包括启动，停止，挂起或重置映像以及获取，删除和还原快照的功能。如果检测到故障或无法到达target，Sulley可以联系此代理
test sequence honing tool will heavily rely on this agent to accomplish its task of identifying the exact sequence of test cases that trigger any given complex fault. ）。

```
ERR> USAGE: vmcontrol.py
<-x|--vmx FILENAME> path to VMX to control
<-r|--vmrun FILENAME> path to vmrun.exe
[-s|--snapshot NAME> set the snapshot name
[-l|--log_level LEVEL] log level (default 1), increase for more verbosity
```

Web Monitoring Interface

Sulle
Session类有一个内置小型Web服务，它被硬编码绑定到端口26000。一旦调用了session的fuzz（ ）方法，Web服务线程就会开始，并且可以看到fuzz的进度，下图是屏幕



通过按下相应按钮可以暂停和恢复fuzzer。 每个检测到的crash在列表中简要显示，其中第一列中列出了违规测试案例编号。
单击测试用例编号可以看到发生crash时加载详细的crash信息。 此信息当然也可以在“crash bin”文件中找到，并可通过编程方式访问。

Post Mortem

一旦Sulley fuzz session完成，就可以查看结果并进入检验阶段。
在Web服务中的会话将为您提供有关潜在未发现问题的早期提示，这是您花时间来分类测试结果。有一实用程序可以帮助您完成此过程。
第一个是'crashbin_explorer.py'实用程序，它接受以下命令行参数：

```
$ ./utils/crashbin_explorer.py
USAGE: crashbin_explorer.py <xxx.crashbin>
[-t|--test #] dump the crash synopsis for a specific test case number
[-g|--graph name] generate a graph of all crash paths, save to 'name'.udg
```

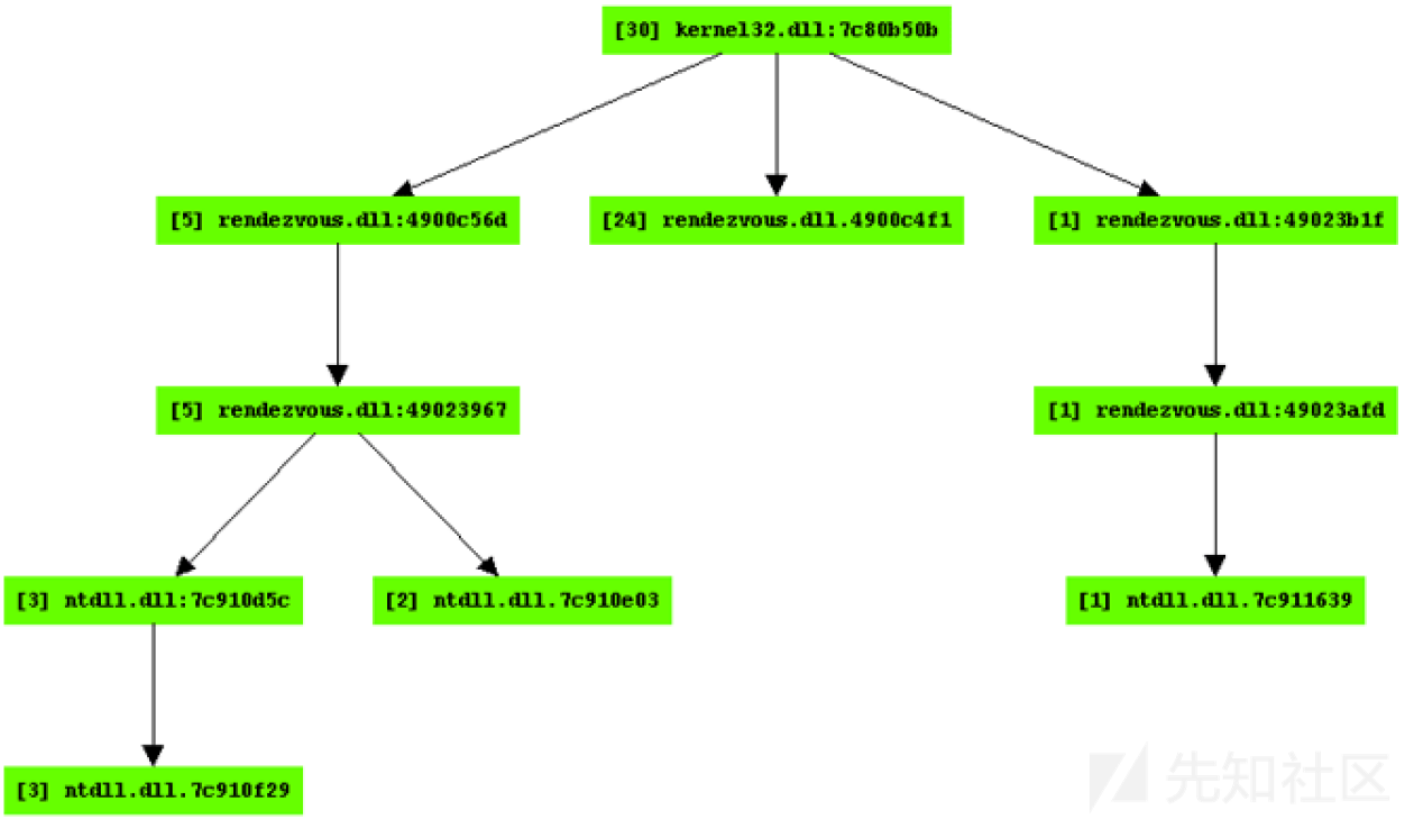
例如，我们可以使用此实用程序查看检测到崩溃的每个位置，并列出在该地址触发崩溃的各个测试用例编号。 以下是对Trillians Jabber协议解析器的真实例子：

```
$ ./utils/crashbin_explorer.py audits/trillian_jabber.crashbin
[3] ntdll.dll:7c910f29 mov ecx,[ecx] from thread 664 caused access violation
1415, 1416, 1417,
[2] ntdll.dll:7c910e03 mov [edx],eax from thread 664 caused access violation
3780, 9215,
[24] rendezvous.dll:4900c4f1 rep movsd from thread 664 caused access violation
1418, 1419, 1420, 1421, 1422, 1423, 1424, 1425, 3443, 3781, 3782, 3783, 3784, 3785, 3786, 3787, [1] ntdll.dll:7c911639 mov cl,
3442,
```

这些列出的崩溃点中没有一个时明显可利用的。 我们可以通过使用'-t'命令行开关指定测试用例编号来进一步深入了解单个崩溃的细节。
我们来看看测试用例编号1416：

```
$ ./utils/crashbin_explorer.py audits/trillian_jabber.crashbin -t 1416
ntdll.dll:7c910f29 mov ecx,[ecx] from thread 664 caused access violation
when attempting to read from 0x263b7467
CONTEXT DUMP
EIP: 7c910f29 mov ecx,[ecx]
EAX: 039a0318 ( 60424984) -> gt;>>...>>>>>(heap)
EBX: 02f40000 ( 49545216) -> PP@ (heap)
ECX: 263b7467 ( 641430631) -> N/A
EDX: 263b7467 ( 641430631) -> N/A
EDI: 0399fed0 ( 60423888) -> #e<root><message>>>>>...>>>& (heap)
ESI: 039a0310 ( 60424976) -> gt;>>...>>>>>(heap)
EBP: 03989c38 ( 60333112) -> \|gt;&t]IP"Ix;IXIox@ @x@PP8|p|Hg9I P (stack)
ESP: 03989c2c ( 60333100) -> \|gt;&t]IP"Ix;IXIox@ @x@PP8|p|Hg9I (stack)
+00: 02f40000 ( 49545216) -> PP@ (heap)
+04: 0399fed0 ( 60423888) -> #e<root><message>>>>>...>>>& (heap)
+08: 00000000 ( 0) -> N/A
+0c: 03989d0c ( 60333324) -> Hg9I Pt]I@"ImI,IIpHsoIPnIX{ (stack)
+10: 7c910d5c (2089880924) -> N/A
+14: 02f40000 ( 49545216) -> PP@ (heap)
disasm around:
0x7c910f18 jnz 0x7c910fb0
0x7c910f1e mov ecx,[esi+0xc]
0x7c910f21 lea eax,[esi+0x8]
0x7c910f24 mov edx,[eax]
0x7c910f26 mov [ebp+0xc],ecx
0x7c910f29 mov ecx,[ecx]
0x7c910f2b cmp ecx,[edx+0x4]
0x7c910f2e mov [ebp+0x14],edx
0x7c910f31 jnz 0x7c911f21
stack unwind:
ntdll.dll:7c910d5c
rendezvous.dll:49023967
rendezvous.dll:4900c56d
kernel32.dll:7c80b50b
SEH unwind:
03989d38 -> ntdll.dll:7c90ee18
0398ffdc -> rendezvous.dll:49025d74
ffffffff -> kernel32.dll:7c8399f3
```

同样，没有明显看出什么东西，但是我们知道因为寄存器包含了ASCII：“&；tg”产生的无效解引用导致了这种访问冲突。或许是字符串扩展问题？我们可以使用“-g”通过图形



我们可以看到，虽然我们已经发现了4个不同崩溃点，但问题的根源似乎是相同的。进一步的研究表明，这确实是正确的。Rendezvous / XMPP（可扩展消息传递和状态协议）消息传递子系统中存在特定缺陷。Trillian通过UDP端口5353上的“_presence”mDNS（多播DNS）服务定位附近的用户。一旦用户通过 \ rendezvous.dll中,跟随逻辑应用于收到的消息(the follow logic is applied to received messages:)：

```
4900C470 str_len:
4900C470 mov cl, [eax] ; *eax = message+1
4900C472 inc eax
4900C473 test cl, cl
4900C475 jnz short str_len
4900C477 sub eax, edx
4900C479 add eax, 128 ; strlen(message+1) + 128
4900C47E push eax
4900C47F call _malloc
```

计算提供的消息的字符串长度，并分配长度为+ 128的堆缓冲区来存储消息的副本，然后通过expatxml.xmlComposeString（）传递

```
seString", struct xml_string_t *);
struct xml_string_t {
unsigned int struct_size;
char *string_buffer;
struct xml_tree_t *xml_tree;
};
```

xmlComposeString（）例程调用expatxml.19002420（），其中HTML分别编码字符&，>和<as&，>和<。 在以下反汇编代码段中可以看到此行为：

```
19002492 push 0
19002494 push 0
19002496 push offset str_Amp ; "&"
1900249B push offset ampersand ; "&"
190024A0 push eax
190024A1 call sub_190023A0
190024A6 push 0
190024A8 push 0
190024AA push offset str_Lt ; "<"
190024AF push offset less_than ; "<"
190024B4 push eax
190024B5 call sub_190023A0
190024BA push
190024BC push
190024BE push offset str_Gt ; ">"
190024C3 push offset greater_than ; ">"
```

```
190024C8 push eax
190024C9 call sub_190023A0
```

由于最初计算的字符串长度不考虑此字符串扩展，rendezvous.dll中的以下后续内联内存复制操作，可能会触发可利用的内存损坏：

```
4900C4EC mov ecx, eax
4900C4EE shr ecx, 2
4900C4F1 rep movsd
4900C4F3 mov ecx, eax
4900C4F5 and ecx, 3
4900C4F8 rep movsb
```

WikiStart
Sulley检测到的每个崩溃都是对这个逻辑错误的回应。 跟踪故障位置和路径使我们能够快速假设崩溃来源。
我们最后一步是希望删除所有不包含崩溃信息的PCAP文件。 'pcap_cleaner.py'实用程序就是为此编写的：

```
$ ./utils/pcap_cleaner.py
USAGE: pcap_cleaner.py <xxx.crashbin> <path to pcaps>
```

此实用程序将打开指定的crashbin文件，读入触发故障的测试用例编号列表并从指定目录中清除所有其他PCAP文件。

上一篇：三次有趣的漏洞跟踪分析 下一篇：利用循环神经网络检测Web攻击
1. 0 条回复
   - 动动手指，沙发就是你的了！