Fuzzing Sucks!

Introducing Sulley Fuzzing Framework

Pedram Amini [1] Aaron Portnoy [2]

[1] pamini@tippingpoint.com

[2] aportnoy@tippingpoint.com

Black Hat US 2007

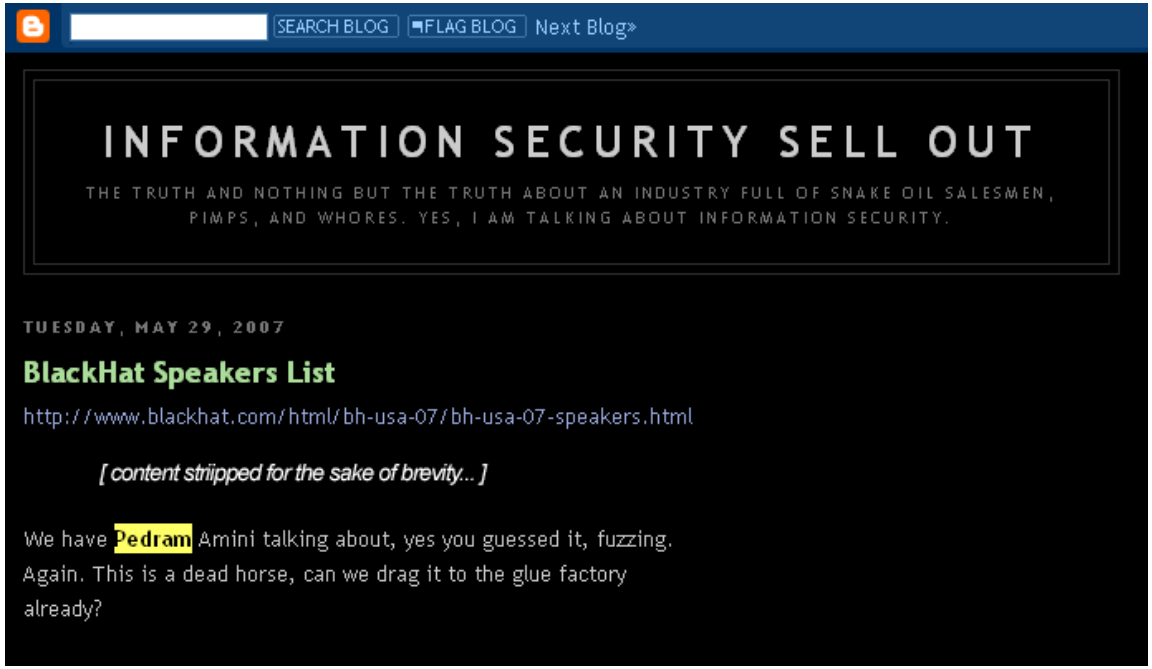
About Us

- Work at TippingPoint's Digital Vaccine Labs
  - Responsible for vuln-dev, patch analysis, pen-testing
  - Keep tabs on us at http://dvlabs.tippingpoint.com
- Launched OpenRCE.org over two years ago
  - How many here are members?
  - Some interesting updates on the horizon after BlackHat
- Creators of PaiMei RE framework
  - How many here have heard of it?
  - Lot of exciting developments coming up after BlackHat
- Co-authored "Fuzzing: Brute Force Vulnerability Discovery"

Talk Outline

- Background
  - Why does fuzzing suck?
  - How can we make it better?
- Sulley's Architecture
  - Component Breakdown
  - Advanced Features
- Usage and Walkthrough
  - Hewlett-Packard Data Protector Audit
  - Trend Micro Server Protect Audit
- Future Development
  - What's still on the drawing board

Is Fuzzing a "Dead Horse"?



Negative

Entire BlackHat track, 3 dedicated books, more commercial vendors and still highly effective.

Old School

- antiparser
  - David McKinney, Python, x-platform, API driven
- DFUZ
  - Diego Bauche, custom language, Unix
- SPIKE
  - Dave Aitel, C, Unix, block based
- The list goes on ...
  - Angel
  - Fuzzer Framework
  - Fuzzled
  - Fuzzy Packet
  - The Art of Fuzzing
  - SPIKEfile
  - ...

DFUZ FTP Example

Notes

The custom language is easy to understand but very limiting.

```
port=21/tcp

peer write: @ftp:user("user")
peer read
peer write: @ftp:pass("pass")
peer read
peer write: "CWD /", %random:data(1024,alphanum), 0x0a
peer read
peer write: @ftp:quit()
peer read

repeat=1024
wait=1
# No Options
```

SPIKE FTP Example

Notes

SPIKE data representation syntax is very simple, perhaps why it's the most commonly used fuzzer?

```
s_string("HOST ");
s_string_variable("10.20.30.40");
s_string("\r\n");

s_string_variable("USER");
s_string(" ");
s_string_variable("bob");
s_string("\r\n");
s_string("PASS ");
s_string_variable("bob");
s_string("\r\n");

s_string("SITE ");
s_string_variable("SEDV");
s_string("\r\n");

s_string("CWD ");
s_string_variable(".");
s_string("\r\n");
```

New School

- Peach
  - Michael Eddington, Python, x-platform, highly modularized
- Codenomicon
  - Commercial vendor, Java, x-platform, pre-recorded test cases
- GPF
  - Jared Demott, mixed, x-platform, varying fuzz modes
- Autodafe
  - Martin Vuagnoux, C, Unix, next-gen SPIKE
  - First fuzzer to bundle debugger functionality
- Evolutionary Fuzzers
  - SideWinder, Sherri Sparks et al.
  - EFS, Jared Demott
- Protocol Informatics Framework
  - Marshall Beddoe, Python, x-platform, automated protocol field identification tool

Peach FTP Example

Notes

There is a non trivial learning curve to writing Peach fuzzers.

```
from Peach              import *
from Peach.Transformers import *
from Peach.Generators   import *
from Peach.Protocols    import *
from Peach.Publishers   import *

loginGroup = group.Group()
loginBlock = block.Block()
loginBlock.setGenerators((
    static.Static("USER username\r\nPASS "),
    dictionary.Dictionary(loginGroup, "dict.txt"),
    static.Static("\r\nQUIT\r\n")
    ))

loginProt = null.NullStdout(ftp.BasicFtp('127.0.0.1', 21), loginBlock)

script.Script(loginProt, loginGroup, 0.25).go()
```

GPF FTP Example

Notes

Data representation format is very different from other examples.

```
Source:S Size:20 Data:220 (vsFTPd 1.1.3)
Source:C Size:12 Data:USER jared
Source:S Size:34 Data:331 Please specify the password.
Source:C Size:12 Data:PASS jared
Source:S Size:33 Data:230 Login successful. Have fun.
Source:C Size:6 Data:QUIT
Source:S Size:14 Data:221 Goodbye.
```

...

The command line can be a bit unwieldy:

```
GPF ftp.gpf client localhost 21 ? TCP 8973987234 100000 0 + 6 6 100 100 5000 43 finsih 0 3 auto none -G b
```

So Why Does Fuzzing Hurt So Bad?

- The existing tools contribute solid ideas but are limited in usage
- Basically all of them are focused solely on data generation
- Let's jump through some fuzzer requirements to get a feel for what's missing
  - Essentially Chapter 5 from the fuzzing book
- At each juncture we'll briefly cover Sulley's solution
- We'll drill down into the specifics when we cover architecture

Easy to Use and Powerfully Flexible

Pain

- Powerful frameworks have a huge learning curve
- Simple frameworks quickly reach limitations

Remedy

- Sulley utilizes block based data representation
- Sulley fuzzers start simple and don't have messy syntax
  - Optional elements and keyword arguments
- Fuzzers are written in pure Python and can benefit from the languages features and ease of use
- Development efforts can be easily shared
- Can handle challenge-response and prev-packet-length situations

Reproducibility and Documentation

Pain

- Individual test cases must be reproducible
- Progress and interim results should be recorded

Remedy

- Sulley can replay individual test cases
- Sulley keeps a bi-directional PCAP of every transaction
- A built in web interface provides interactive feedback

Reusability

Pain

- Non-generic fuzzers can never be used again
- Widely used protocol components are re-developed all the time

Remedy

- Sulley supports the creation and reuse of complex types and helper functions
- The more Sulley is used, the smarter it gets

Process State and Process Depth

Pain

- None of the existing fuzzers consider state paths
  - Fuzzing A-B-D vs. A-C-D
- Many fuzzers can only scratch a protocol surface
  - Fuzzing A only

Remedy

- In Sulley you build fuzzers in manageable chunks called requests
- These requests are tied together in a graph
- The graph is automatically walked and each state path and depth is individually fuzzed

Tracking, Code Coverage and Metrics

Pain

- How much of the target code was exercised?
- What code was executed to handle a specific test case?

Remedy

- Sulley supports an extensible agent model
- Utilizes PaiMei/PyDbg for breakpoint-based and MSR-based code coverage tracking

Fault Detection and Recovery

Pain

- Most fuzzers rely solely on a lack of response to determine when something bad happens
  - ahem ahem Codenomicon
- Once a fault is discovered, most fuzzers simply stop!
  - Mu Security and BreakingPoint take the interesting approach of power cycling

Remedy

- Sulley bundles a debugger monitor agent
- Sulley can restore target health and continue testing by:
  - Restarting the target service
  - Restoring a VMware snapshot

Resource Constraints

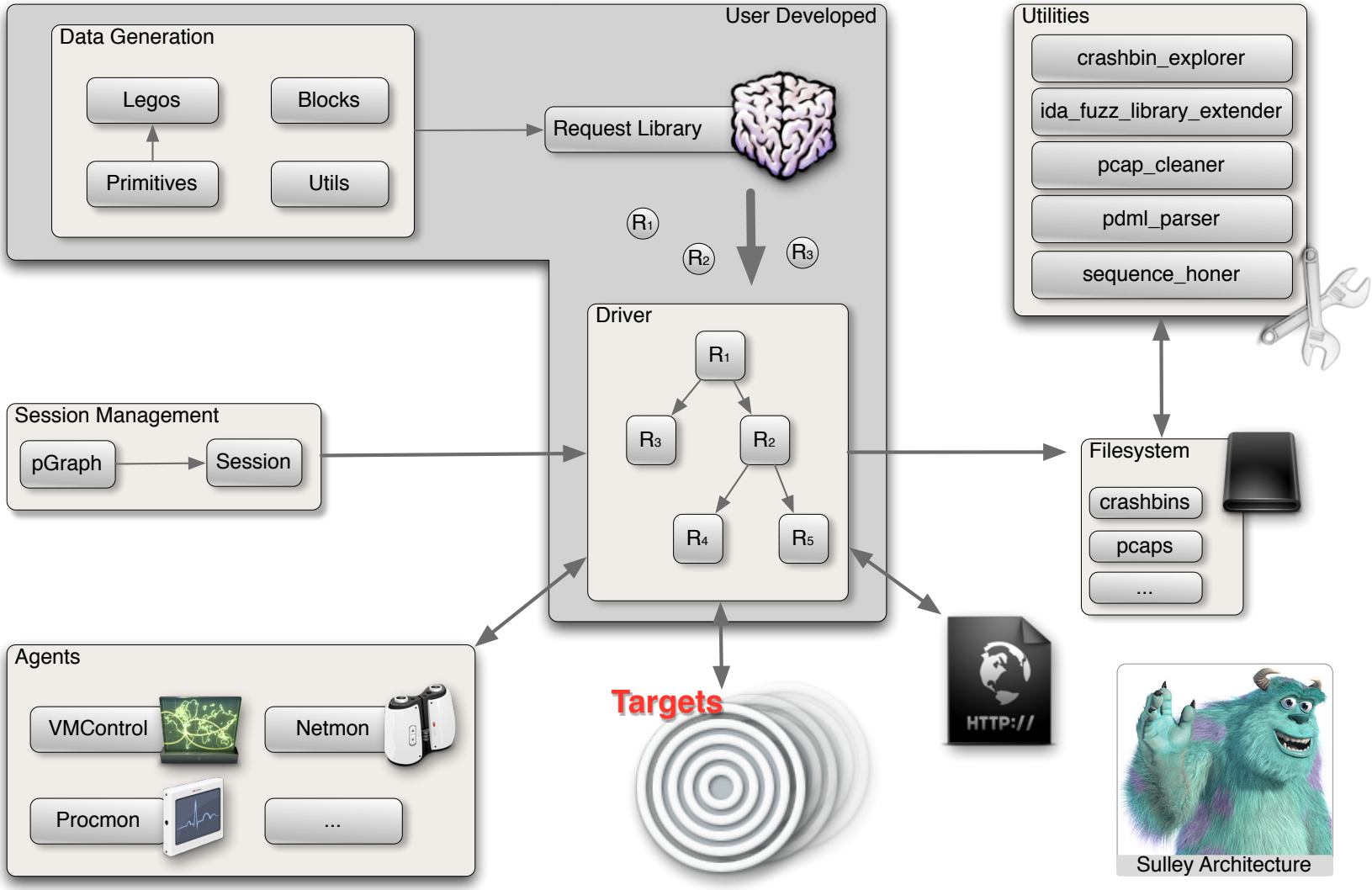
Pain

- Non-technical constraints such as time and man power often get in the way

Remedy

- Sulley bundles utilities such as a PDML parser to save time
- Sulley is designed to allow multiple people to work together easily
- The monitoring and self recording features of the framework save a great deal of time

Sulley Architecture Diagram



Sulley Architecture

Four Major Components

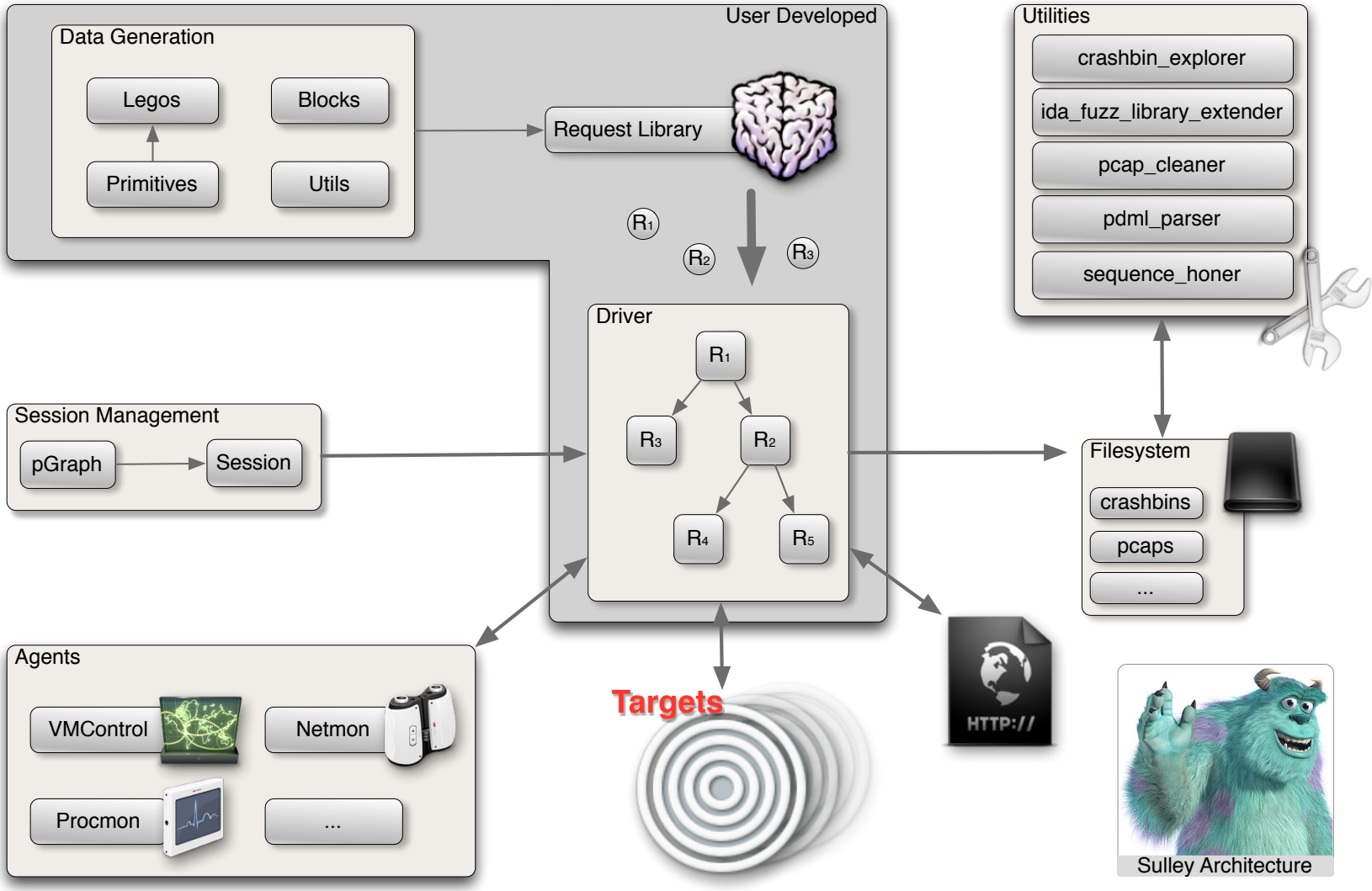
- Data Generation
  - You build requests out of primitives and legos
  - Legos are complex types that extend the framework
- Session Management / Driver
  - Requests are chained together in a graph to form a session
  - The session class exposes a standalone web interface for monitoring and control
  - The driver ties targets, agents and requests together
- Agents
  - Interface with the target for instrumentation and logging purposes
- Utilities
  - Standalone command line utilities that perform a variety of tasks

General Usage

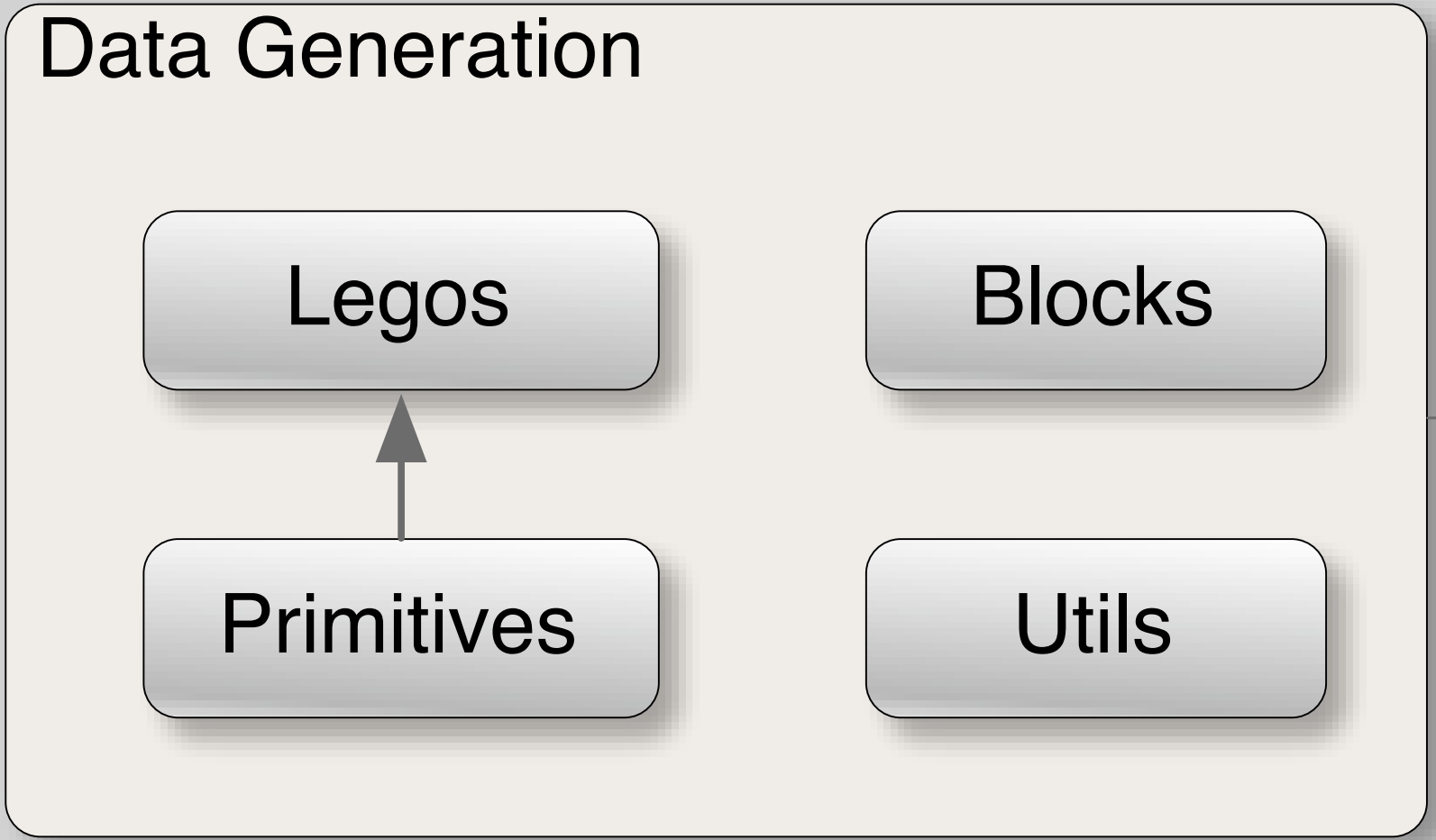
- Sniff some traffic or reverse some protocol parsing binary
- Break the target protocol into individual requests
- Represent each request with a series of primitives
  - Individual requests can be tasked out to different people
- Setup some targets in conjunction with various agents
- Write a driver script which instantiates a session and ties requests, agents and the targets together
- Fuzz!
- Review results

Drill Down

Let's take a look at each individual component in detail...



Sulley Architecture


Data Generation
Legos
Blocks
Primitives
Utils

Overview

- Aitel really had it right
- The block based approach to protocol representation is simple, flexible and powerful
- Protocols are represented in Sulley as a collection of primitives, blocks and block helpers
- These elements have many optional arguments
- The name optional argument gives you direct access to an element without having to walk the stack
- Refer to the Epydoc generation API documentation for a complete reference of optional arguments
- Begin the definition of a request with:
  - `s_initialize("request name")`

Static and Random Primitives

- s_static() is the simplest primitive adding a constant to the stack
  - Aliases include s_dunno(), s_raw() and s_unknown()
- s_binary() is related and should be familiar to SPIKE users:
  - `s_binary("0xde 0xad be ef \xca fe 00 01 02 0xba0xdd")`
- Sulley primitives are driven by heuristics, with the exception of s_random()
- s_random() used to generate random data of varying lengths
  - min_length and max_length are mandatory arguments
  - num_mutations defaults to 25 and specifies how many values to cycle through prior to returning to default

Integer Primitives

- Simple types for dealing with integer fields
  - s_char(), s_short(), s_long(), s_double()
  - Convenience aliases exist like byte, word, dword, int, etc...
- You can fuzz through the entire valid range
- Defaults to a subset of potentially interesting values
  - To increase throughput
- Supports ASCII (signed or unsigned) and binary output rendering

Strings and Delimiters

- s_string() supports static sizes, variable padding and custom
  - Over 1,000 test cases in string fuzz library
- Strings are frequently parsed into sub-fields with delimiters
- Sulley has a special primitive for delimiters, s_delim(), here is an example:

```
# fuzzes the string: <BODY bgcolor="black">
s_delim("<")
s_string("BODY")
s_delim(" ")
s_string("bgcolor")
s_delim("=")
s_delim("\"")
s_string("black")
s_delim("\"")
s_delim(">")
```

Blocks

- Primitives (and blocks) can be organized and nested within blocks
- Blocks are opened and closed with s_block_start() and s_block_end() respectively
- Blocks can be associated with a group, encoder or dependency
- Grouping, encoding and dependencies are powerful features we examine individually
- s_block_start() returns True so you can tab out for readability:

```
# Blocks must be given a name.
if s_block_start("my block"):
    s_string("fuzzy")
s_block_end()
```

Groups

- Groups tie a block to a defined set of values
- The block is cycled through for each value in the group
- Useful for representing a valid list of opcodes or verbs:

```
# define a group primitive listing the various HTTP verbs we wish to fuzz.
s_group("verbs", values=["GET", "HEAD", "POST", "TRACE"])

# define a new block named "body" and associate with the above group.
if s_block_start("body", group="verbs"):
    s_delim(" ")
    s_delim("/")
    s_string("index.html")
    s_delim(" ")
    s_string("HTTP")
    s_delim("/")
    s_string("1")
    s_delim(".")
    s_string("1")
    s_static("\r\n\r\n")
s_block_end("body")
```

Encoders

- Simple yet powerful block modifier
- Connect a function with a block to modify contents post render
- Implement compression, custom obfuscation, etc:

```
def trend_xor_encode (str):
    key = 0xA8534344
    pad = 4 - (len(str) % 4)

    if pad == 4: pad = 0
    str    += "\x00" * pad

    while str:
        dword  = struct.unpack("<L", str[:4])[0]
        str    = str[4:]
        dword ^= key
        ret   += struct.pack("<L", dword)
        key    = dword
    return ret
```

Dependencies

- Allow you to apply a conditional to the rendering of a block
- Done by specifying dep, dep_value(s) and dep_compare
- Block dependencies can be chained together arbitrarily

```
s_short("opcode", full_range=True)

if s_block_start("auth", dep="opcode", dep_value=10):
    s_string("aaron")
s_block_end()

if s_block_start("hostname", dep="opcode", dep_values=[15, 16]):
    s_string("aportnoy.openrce.org")
s_block_end()

# the rest of the opcodes take a string prefixed with two underscores.
if s_block_start("something", dep="opcode", dep_values=[10, 15, 16], dep_compare="!="):
    s_static("__")
    s_int(10)
s_block_end()
```

Sizers

- Use s_size() to dynamically measure and render a blocks size
- Many optional arguments for flexibility
  - length of size field, default is 4
  - endianess of size field, default is little
  - Output format, "binary" (default) or "ascii"
  - inclusive, whether the sizer should count itself
  - With ASCII output control signed vs. unsigned
- Sizers can also be fuzzed

Checksums

- Similar to a sizer, s_checksum() calculates and render the checksum for a block
- Keyword argument algorithm can be one of:
  - "crc32"
  - "adler32"
  - "md5"
  - "sha1"
  - or any arbitrary function pointer
- endianness can be toggled

Repeaters

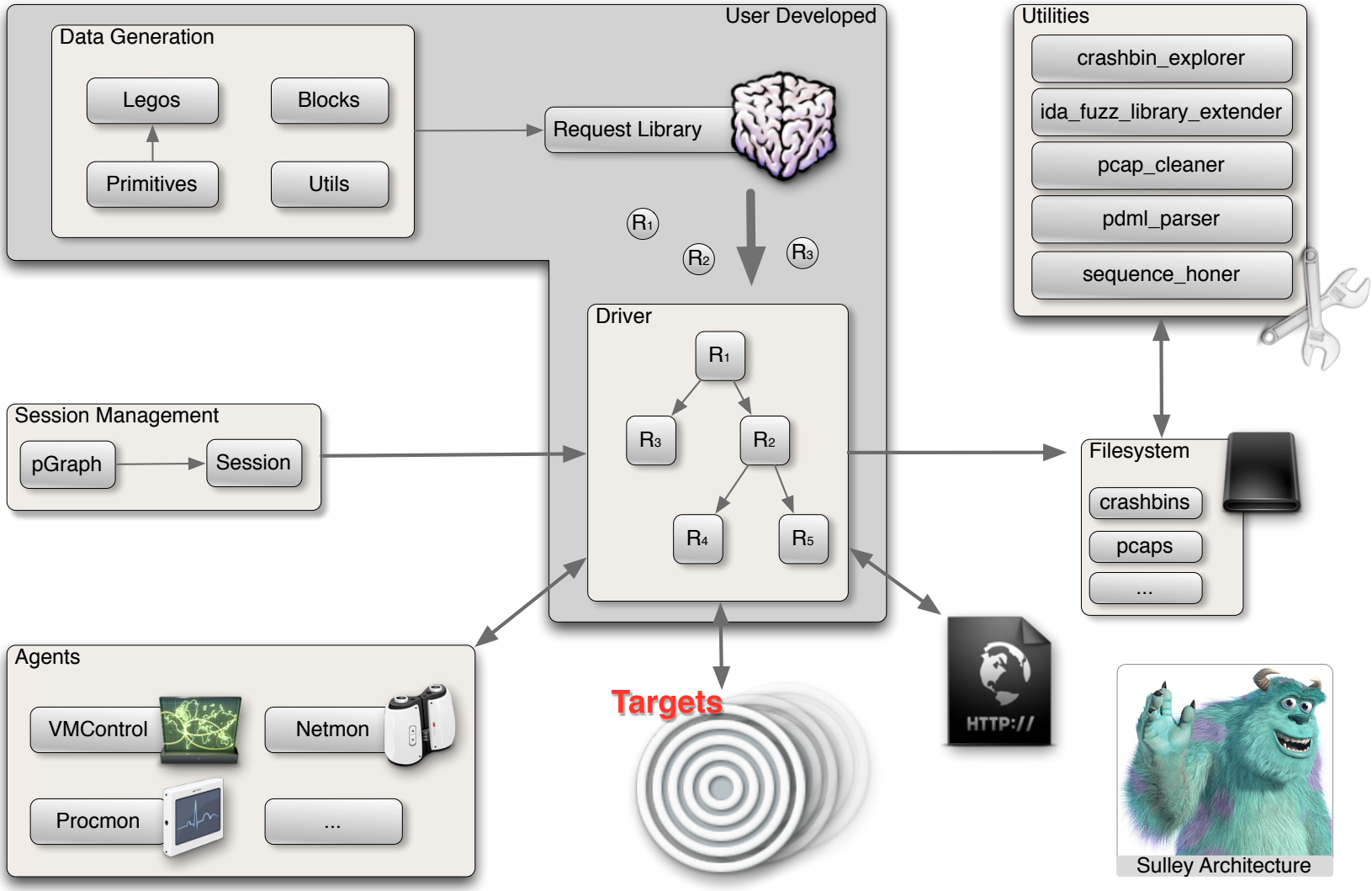
- Use s_repeat() to handle block repetitions
- Useful for fuzzing multi-entry table parsers
- Can variable step from min_reps to max_reps
- Alternatively the repeat factor can be tied to another variable

```
# table entry: [type][len][string]
if s_block_start("table entry"):
    s_random("\x00\x00", 2, 2)
    s_size("string field", length=2)

    if s_block_start("string field"):
        s_string("C" * 10)
    s_block_end()
s_block_end()

# repeat the table entry from 100 to 1,000 reps stepping 50 elements on each iteration.
s_repeat("table entry", min_reps=100, max_reps=1000, step=50)
```

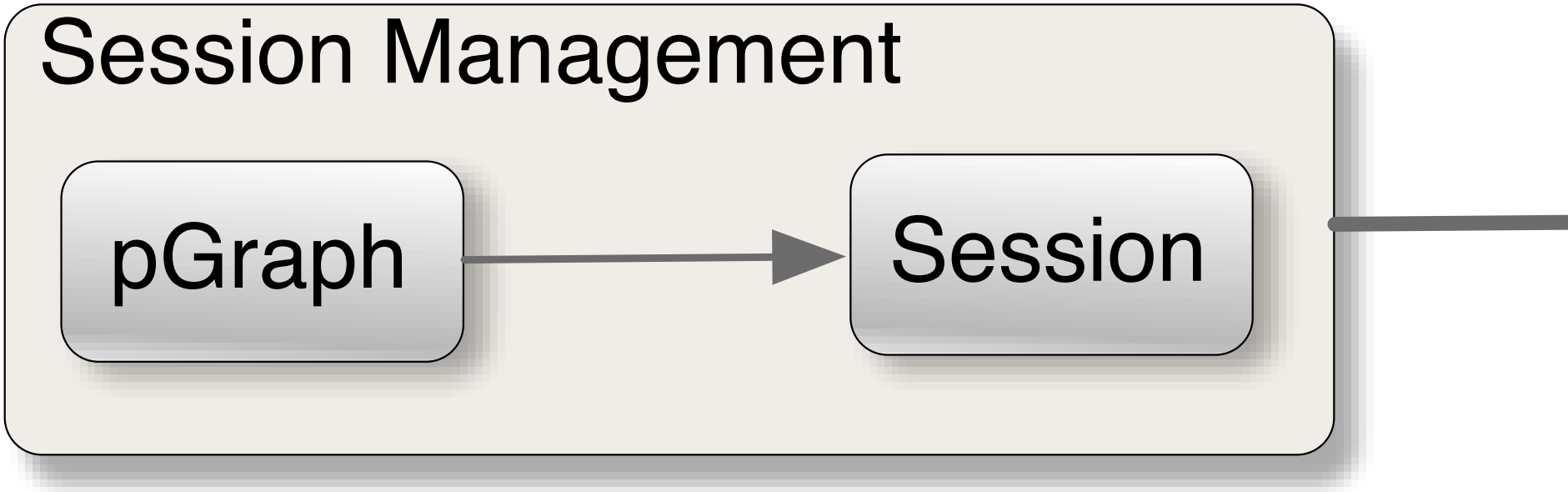
Legos

- Sulley supports the creation of complex types called Legos
- Example Legos include: E-mail addresses, IP addresses, DCERPC, XDR and ASN.1 / BER primitives, XML tags, etc...
- The more Legos you define, the easier fuzzing is in the future

```
class tag (blocks.block):
    def __init__ (self, name, request, value, options=):
        blocks.block.__init__(self, name, request, None, None, None, None)

        self.value   = value
        self.options = options

        # [delim][string][delim]
        self.push(primitives.delim("<"))
        self.push(primitives.string(self.value))
        self.push(primitives.delim(">"))

# example instantiation.
s_lego("tag", "center")
```



Sulley Architecture


Session Management
pGraph
Session

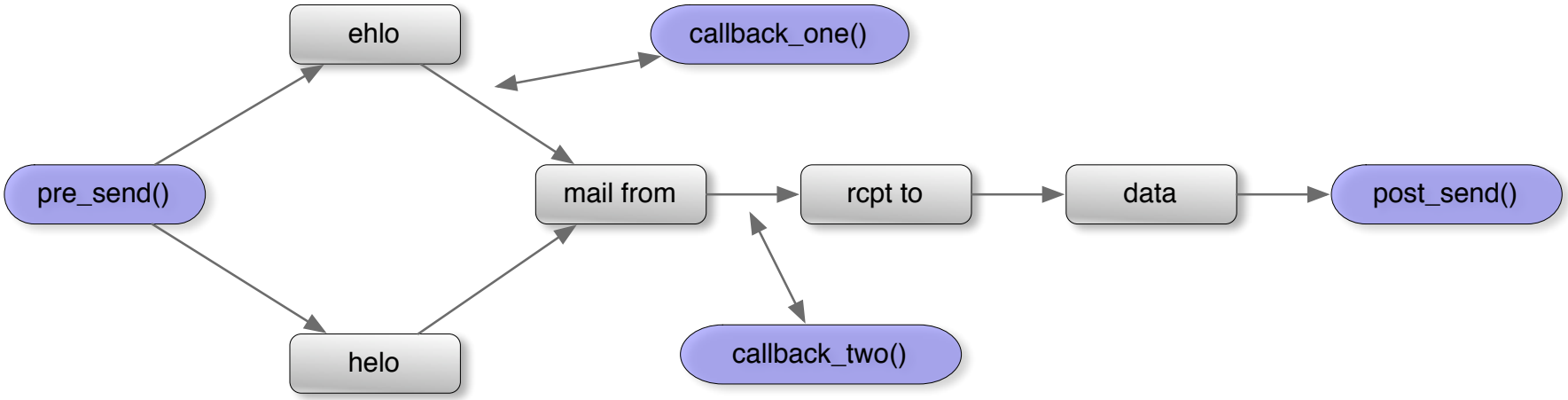
pGraph

- Python graph abstraction library
- Developed originally for PaiMei
- Allows for simple graph construction, manipulation and rendering
- Rendering formats supported are GML, GraphViz and uDraw
- The session class extends from this...

Session Class

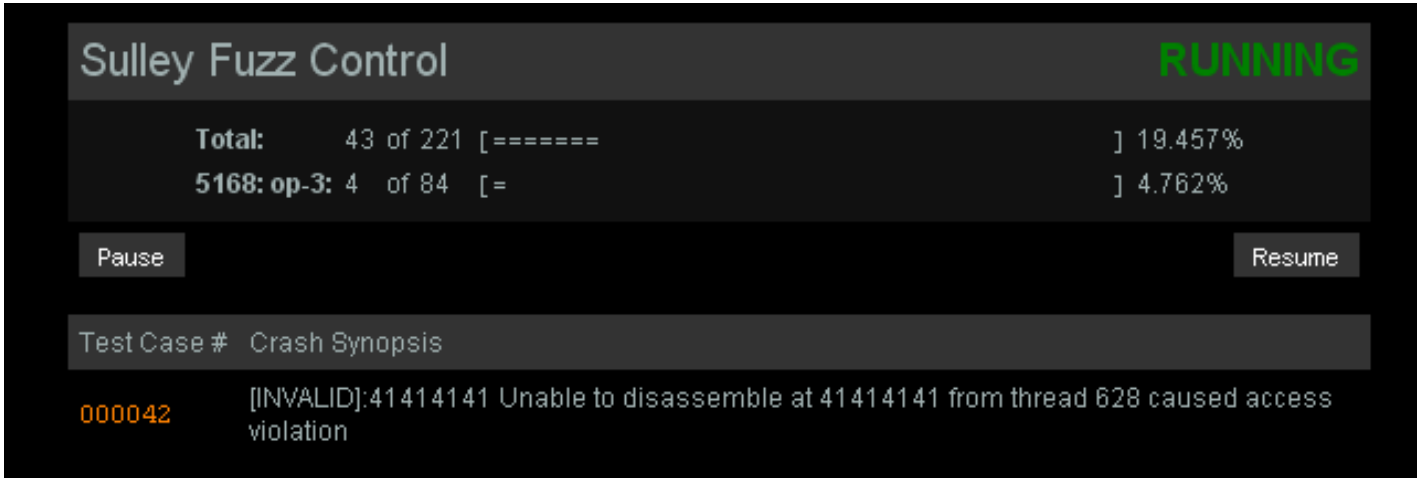
- Connect multiple requests in a graph
- Register pre and post send callbacks
- Assign a callback to each edge
- Add multiple network targets
- Exposes a custom web interface
- Automatically communicates with registered agents
- Responsible for walking the graph and fuzzing at each level
- Tightly used and related to the creation of drivers

Session Illustrated

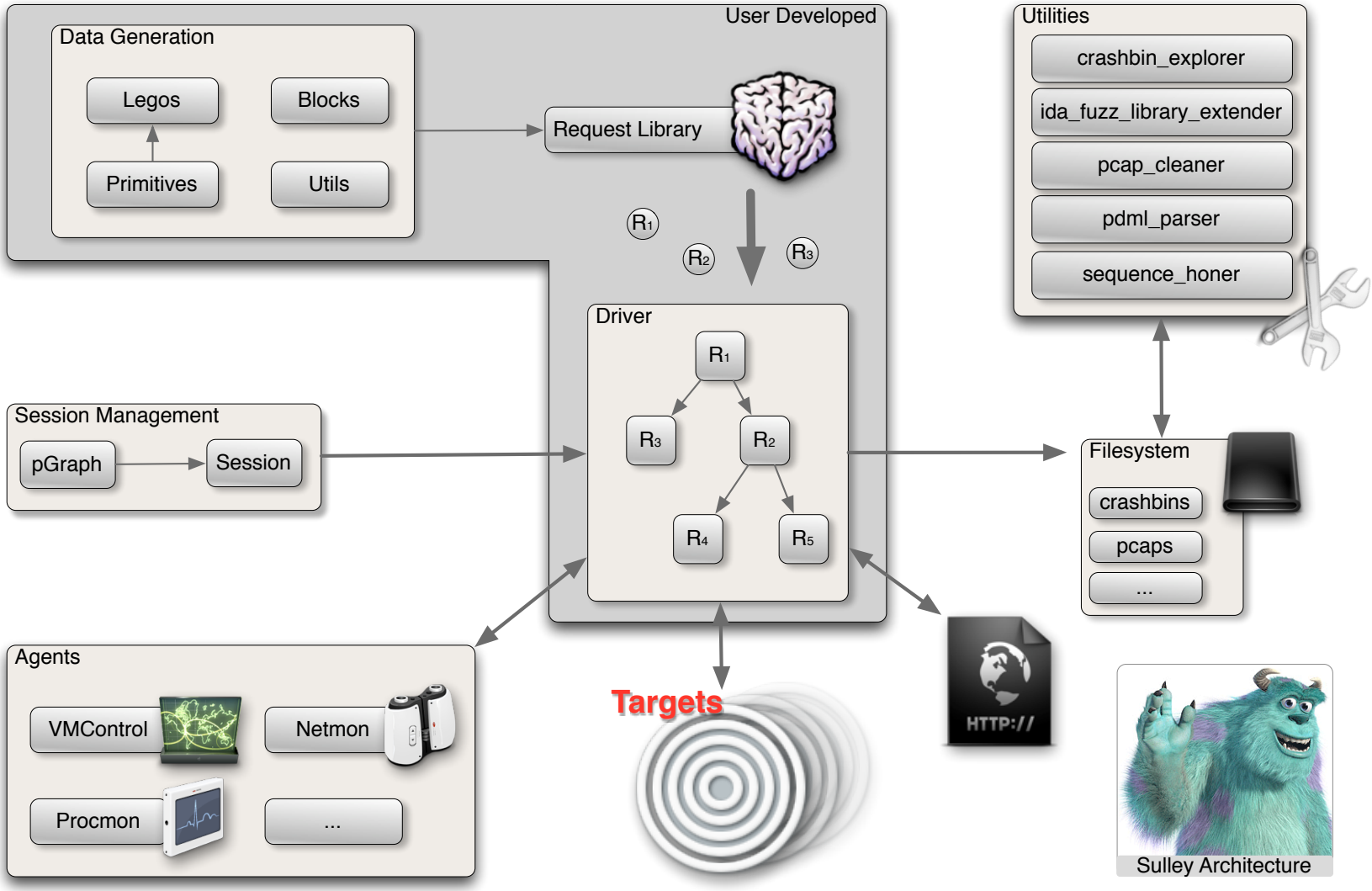


- This example demonstrates multiple paths and depths
- Callbacks aren't really needed here
  - More applicable in cases of RPC, challenge-response, prev-packet specifies length, etc...

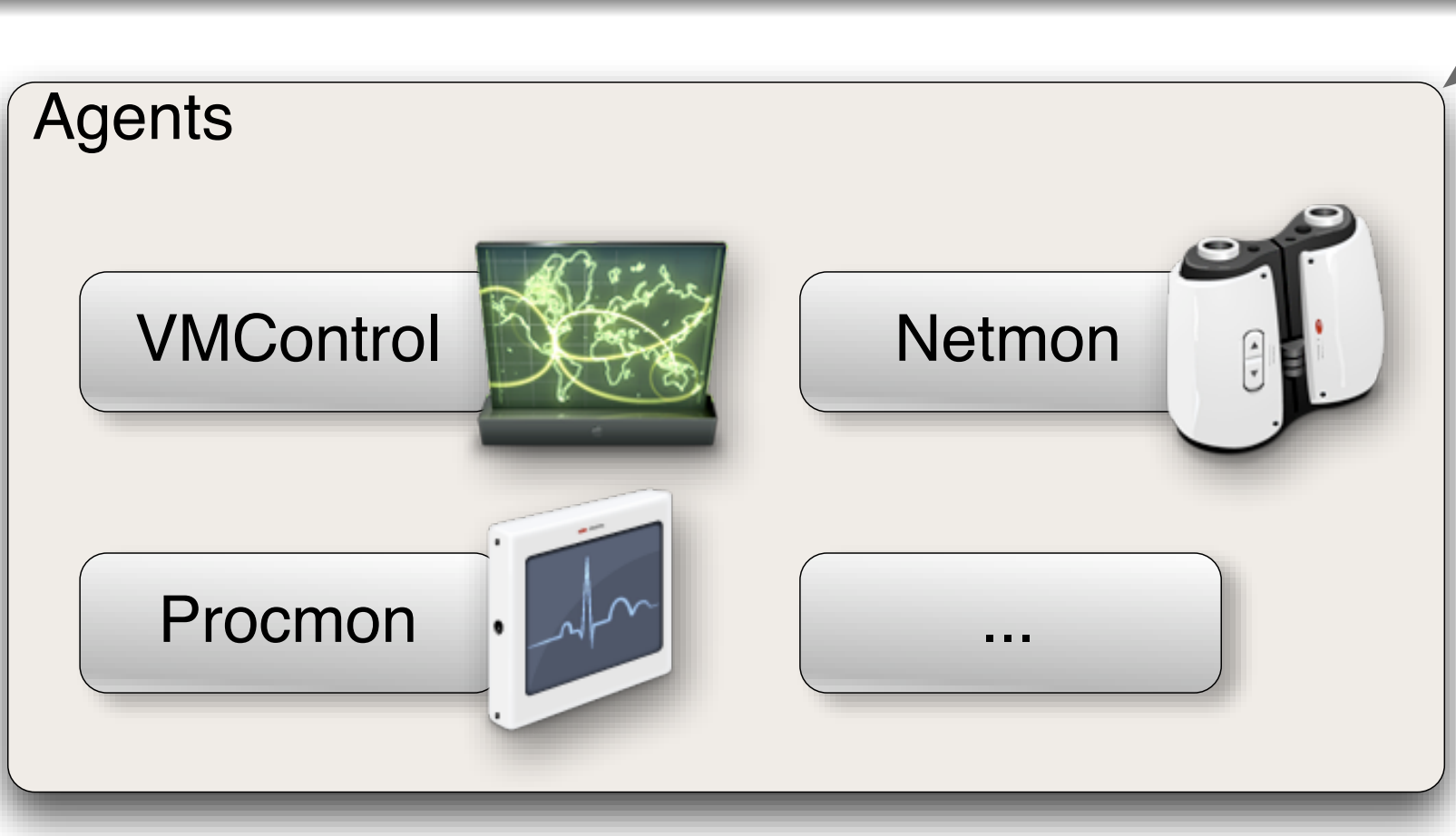
Interactive Web Interface



- View fuzzer progress
- View detected faults
- Retrieve per-fault crash dump and packet capture
- Pause and resume fuzzing



Sulley Architecture


Agents
VMControl
Netmon
Procmon
...

Agents

- A flexible sub-system allows you to create custom agents
- Client-server communication is extremely simple over "PedRPC"
  - Create a class that extend from pedrpc.server
  - Instantiate pedrpc.client
  - Call class members as if they are local
- Some agents have already been developed...

Agent: Netmon

- Monitors network traffic and saves PCAPs to disk
- Per test case bi-directional packet capture

```
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
    [--port PORT]              TCP port to bind this agent to

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: Procmon

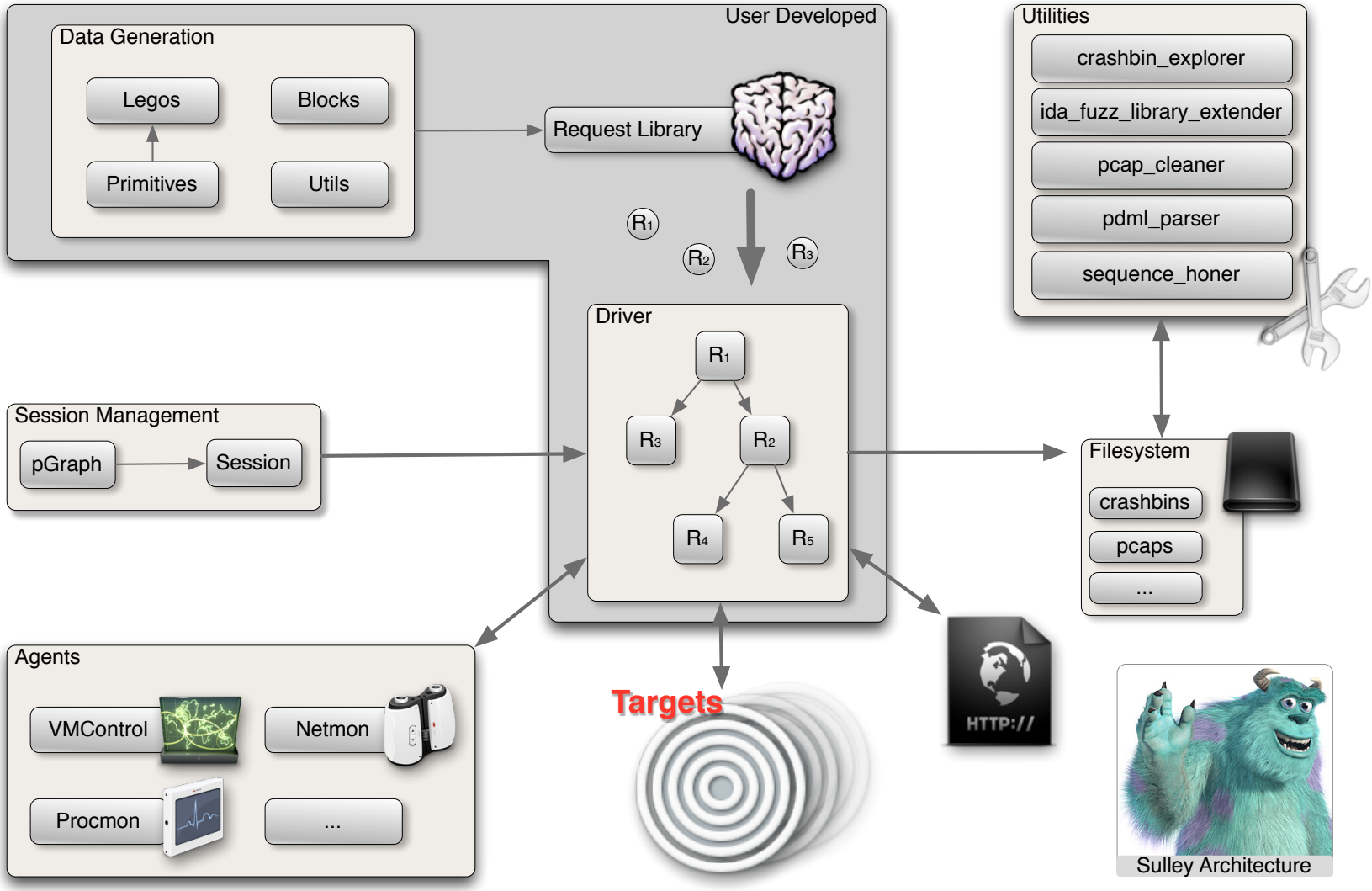
- PyDbg based fault monitoring engine
- Used to monitor target health
- Detected faults are catalogued in a "crash bin"
  - Allows for simplistic (backtrace driven) fault clustering
  - More on this later

```
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
    [--port PORT]              TCP port to bind this agent to
```

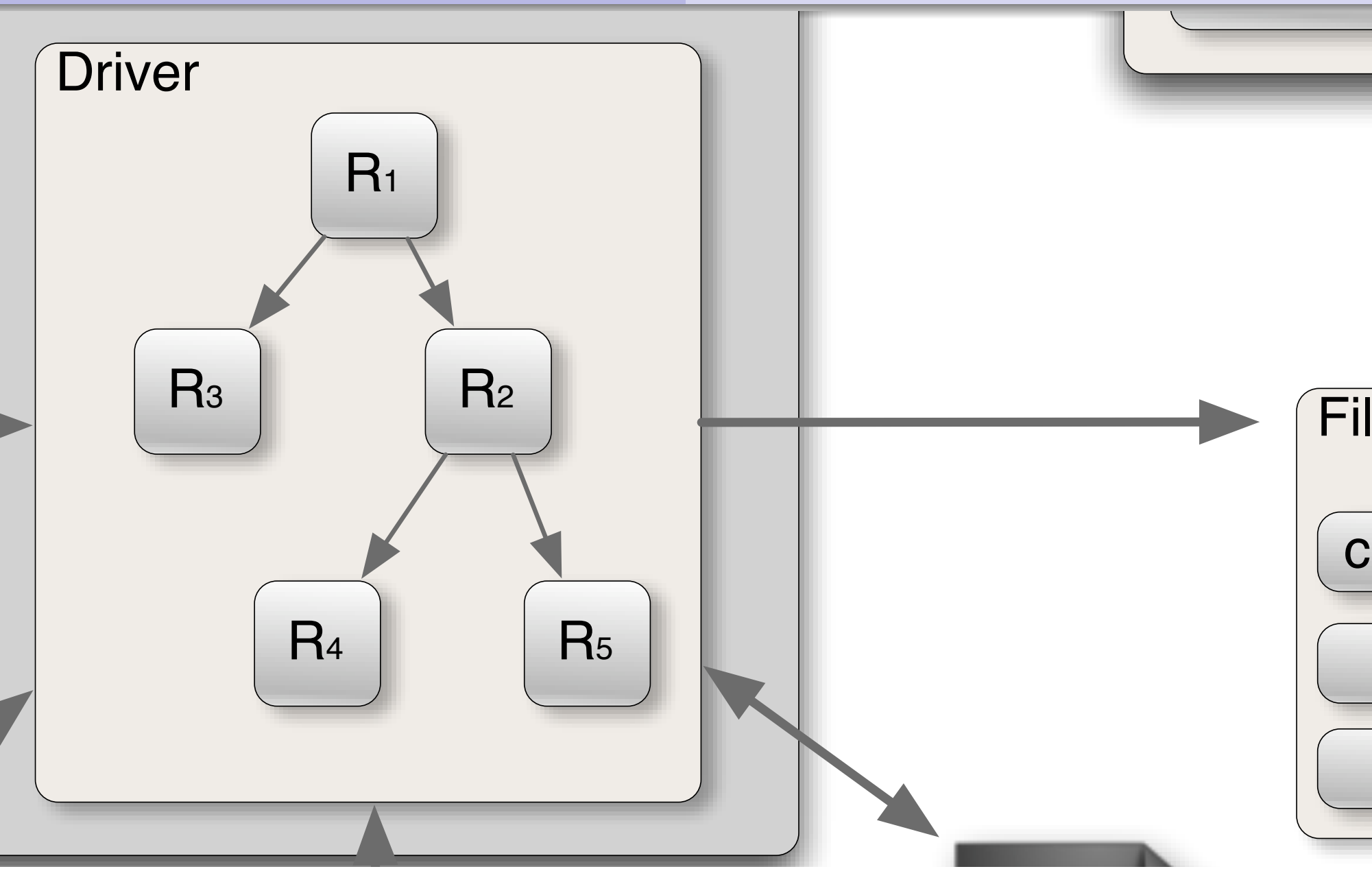
Agent: VMControl

- Exposes network API for VMWare instrumentation
  - Simple PedRPC wrapper around vmrun.exe
  - Start, stop, suspend, snapshot, revert, etc...
- Used to restore target health after a fault is induced

```
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
    [-i|--interactive]      Interactive mode, prompts for input values
    [--port PORT]           TCP port to bind this agent to
```



Sulley Architecture

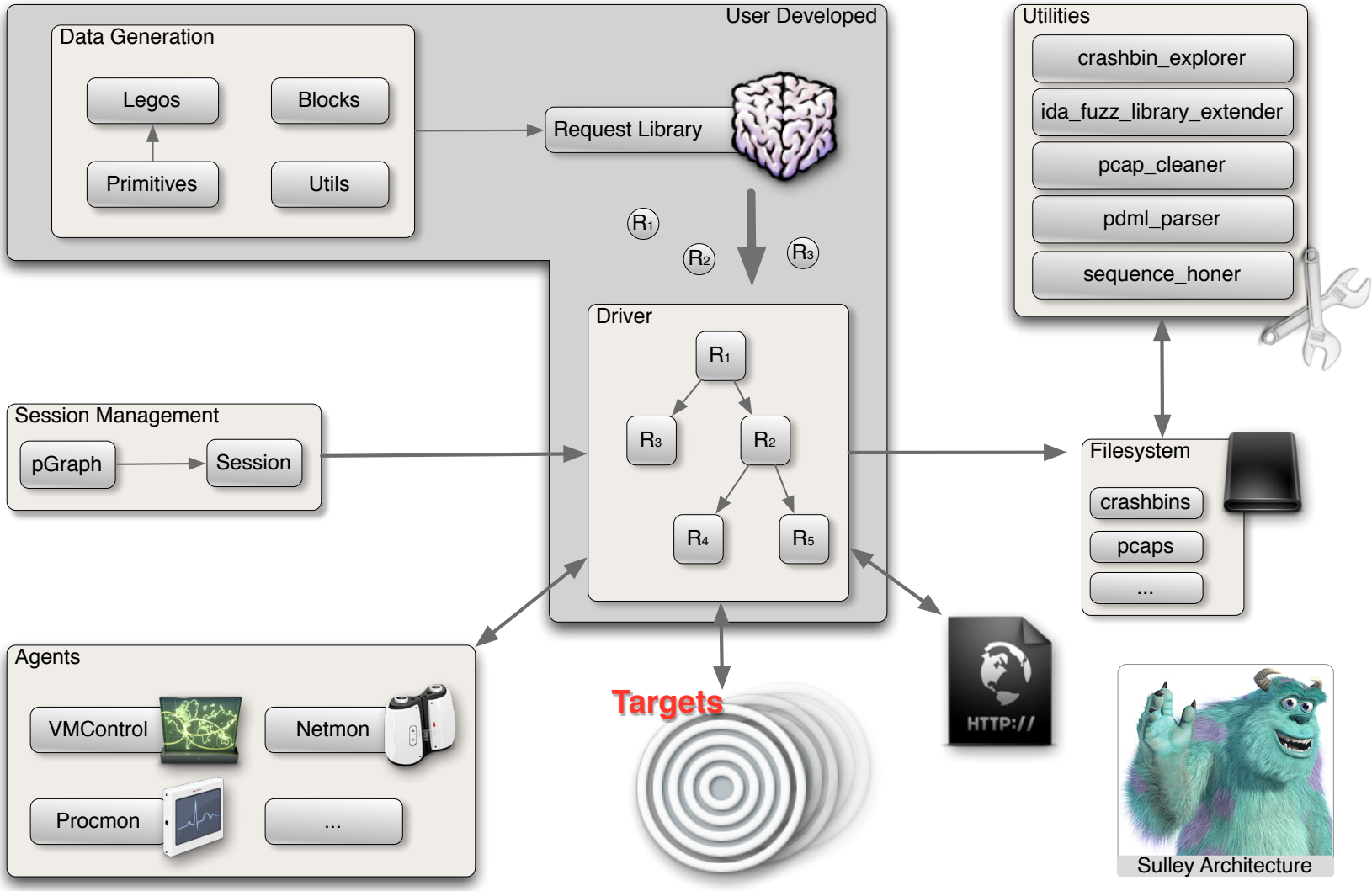

Driver
R1
R3
R2
R4
R5
Fil
c

Driver

- The driver is where it all comes together:
  - Import requests from the request library
  - Instantiate a session instance
  - Instantiate and add target instances to the session
  - Interconnect the requests to form a graph
  - Start fuzzing
- This is where edge and pre/post send callbacks should be defined
- The driver is entirely free form, though must of them will follow a simple and similar structure

Example Driver

```
from sulley   import *
from requests import jabber

def init_message (sock):
    init  = '<?xml version="1.0" encoding="UTF-8" ?>\n'
    init += '<stream:stream to="10.10.20.16" xmlns="jabber:client" xmlns:stream="http://etherx.jabber.org">'

    sock.send(init)
    sock.recv(1024)

sess             = sessions.session(session_filename="audits/trillian.session")
target           = sessions.target("10.10.20.16", 5298)
target.netmon    = pedrpc.client("10.10.20.16",   26001)
target.procmon   = pedrpc.client("10.10.20.16",   26002)
target.vmcontrol = pedrpc.client("127.0.0.1",     26003)

# start up the target.
target.vmcontrol.restart_target()
print "virtual machine up and running"

sess.add_target(target)
sess.pre_send = init_message
sess.connect(s_get("chat message"))
sess.fuzz()
```



Sulley Architecture

Utilities

- crashbin_explorer
- ida_fuzz_library_extender
- pcap_cleaner
- pdml_parser
- sequence_honer

crashbin_explorer.py

- View every test case which caused a fault
- List every location where a fault occurred
- Retrieve saved crash dumps
- Render a graph which clusters faults by stack trace

```
$ ./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
```

- Pedram will actually demo this in a bit

ida_fuzz_library_extender.py

- IDA Python script
- Simple concept
  - Enumerate all constant integer comparisons in target binary
  - Enumerate all constant string comparisons in target binary
  - Add to fuzz library heuristics
- This is a pre-fuzz static analysis script
- A more advanced run-time implementation is still in the works
- Fuzz library extensions are handled via .fuzz_strings and .fuzz_ints

pcap_cleaner.py

- Simple utility
- Iterates through a crashbin and removes any PCAPs not associated with a fault
- Save on disk space prior to archiving a completed audit

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

pdml_parser.py

- Convenience utility
- Converts PDML dump from Wireshark to a Sulley request
- Easier then doing so manually, work is still required of course

Hewlett-Packard Data Protector

- Simple protocol to reverse and represent
- Simple bug (currently 0day, fix is just around the corner)
- Good starting example

Trend Micro ServerProtect

- Microsoft RPC interface
- Tons of bugs in this thing
- Some have been reported and fixed, others are still pending
- Interesting demo since we can show off the fact that Sulley can fuzz DCE/RPC

Sequence "Honing"

- Say test case #100 triggered a fault but replaying it does not
- Probably relies on some previous test or sequence of tests
- We can automatically deduce the exact sequence required for replication
- This is done, but I'm still playing with the reduction "algorithm"
- Here is how the current incarnation of the approach works...

The Honing "Algorithm"

- Start from the last case and step back to find the window
  - 100. 99, 100. 98, 99, 100. 40, 41 ... 99, 100
- Start eliminating sequential bundles (ratio to window-size) and check for fault
- Once exhausted, increase granularity to eliminate single test cases at a time
- Sulleys health-restoration methods are used to ensure a "clean slate" per test

Parallel Fuzzing

- Combinatorial explosion is a common fuzzing problem
- We can increase throughput by replicating the target, ex:
  - Run 2 targets in VMWare with PyDbg monitoring
  - Run a target on real hardware for MSR-based code coverage recording
- Parallel fuzzing is as simple as instantiating and add multiple targets in your driver code

Heuristic Feedback Loop

- Along the same lines as the fuzz extender command-line utility
- Simple concept that may improve your fuzz:
  - Attach to the target process and trace
  - Look for all comparisons to int and string constants
  - Feed those back to the fuzzer, adding them to the current primitives fuzz library
- There are far more scientific ways of doing this
  - Hoglund and Halvar both are researching what I call "path-math"
  - Was talking to a friend at Microsoft, they already have it

Code Coverage Monitor

- An agent similar to netmon or procmon, records code coverage per test case
- There are 2 code coverage monitors completed
  - MSR based
  - Process Stalker based
- Neither are included as they add a lot more requirements (PaiMei)
- We're working on PaiMei 2.0 which will be SQL driven and will include it then

PCAP Binning

- Crash binning is great, we can apply it to packets as well
  - Monitor and parse network responses from target
  - Group them together as best we can
  - ex: 404 responses vs. 202 responses from a web server
- Most useful when you are fuzzing a target you can't monitor with other tools
  - Say the Xbox for example

File Fuzzing

- The session class, driver etc. was all designed for network protocol fuzzing
- The data representation stuff however is generic
- Peter Silberman has already written a beta file-session for file fuzzing
- Alternatively, you can...
  - Represent the file in Sulley blocks
  - Write a 3 line .render() loop to generate your test cases
  - Use Cody Pierce's PaiMei FileFuzz module as a testing harness

Web Based GUI

- Coding up blocks is an easy task, but not easy enough for widespread use
- We'd like to have a drag and drop GUI
  - Create and nest blocks visually
  - Insert, re-arrange, etc. primitives
  - View and modify options for each primitive
  - Save it off to a request library
  - Create a session and connect requests
  - Configure and add targets
  - Generate the driver automatically
- We have no GUI skills, someone else needs to do this

Appliance Based Distribution

- Extending on the previous though...
- With a nice GUI we can move to an appliance model
  - Distribute Sulley as a virtual machine
- Simply start it up, configure over web and attack a target
- We can do auto updates, etc...
- Maybe one day

Questions? Comments?

- About the tool
- About ...

Total Slide Count

65