GRAPE : Generative Fuzzing

Fourteenforty Research Institute, Inc.
http://www.fourteenforty.jp

Nicholas Green
Darren Willis, Takahiko Funakubo

Grape

A Generative Fuzzer

- Inspired by Scapy , Sulley, PeachFuzz, et cetera
- Generalized Fuzzing: can fuzz packets, files, higher level interactions
- Handles responses: can interact with stateful protocols

Outline

- What is fuzzing (very briefly)
  - Types of fuzzing
  - Challenges in fuzzing
- Our fuzzer (GRAPE)
  - Overview of a fuzzing 'scenario'
  - How GRAPE specifies its rules
  - How GRAPE handles complex logic (macros)
  - How GRAPE handles statefulness and participates in 'conversations'
- Demo
  - Grape vs Windows 7
  - Grape vs a router

Fuzzing: most basic case

Normal Data →

← Normal response

Abnormal data →

Normal Data →

System
Under
Test

Problem found ← **No normal response**

Fuzzing (Very Much In Brief)

- Testing a system by subjecting it to malformed inputs
- Broadly, two types
  - Mutating - Take existing inputs, tweak them
    - Random Bit Flipping
    - Field alteration (requires knowledge of fuzzed format)
    - Input samples important
  - Generative - Use set of rules to create new inputs
    - Also requires knowledge of fuzzed format
    - Rules determine coverage

Fuzzing steps

- Find or define attack surface
- Generate Input Cases
- Feed Them To Target
- Monitor For Crashes / Unusual Behaviour
- Collect & Analyse Crash Data

Fuzzers - Generality

- Most fuzzers are quite specific
  - Fuzzers for various protocols
    - SNMP/DHCP/ICMP/etc
  - Fuzzers for specific file formats
    - PDF/HTML/SWF/etc

Scapy is an example of a more general fuzzing system, but still network focused.

Fuzzers - Smartness

- Fuzzers vary in ‘randomness’
- Most fuzzers are smart
  - Requires understanding the format of the input being fuzzed
  - Mutate/Generate input such that it’s likely to break the system (length fields, etc)

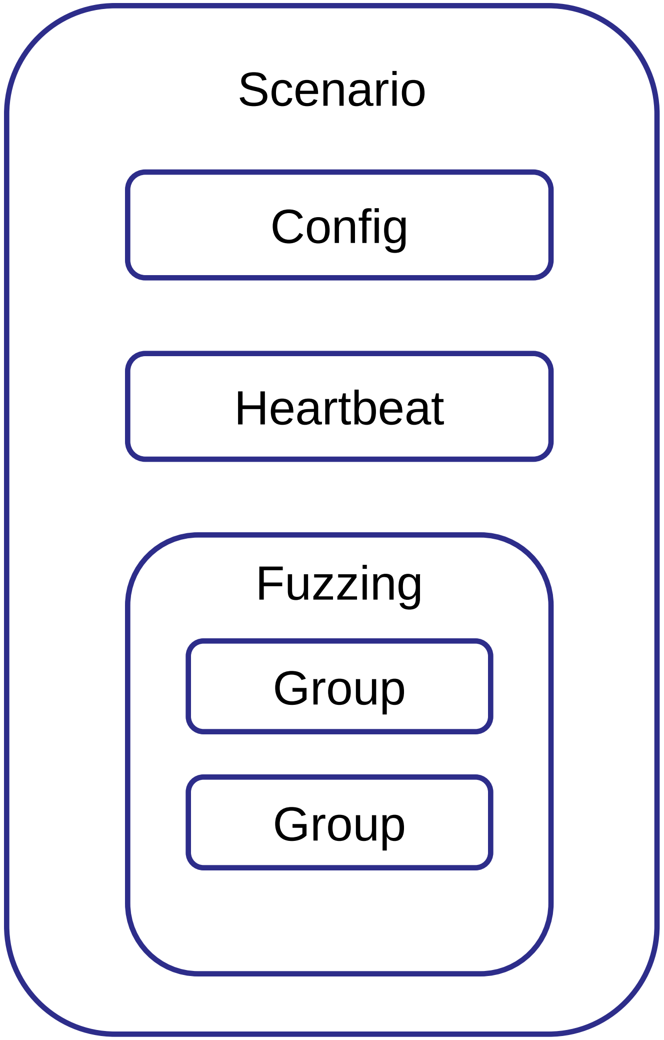
Generally: Try to imagine how someone would have messed up trying to implement the code parsing the input you’re attacking.

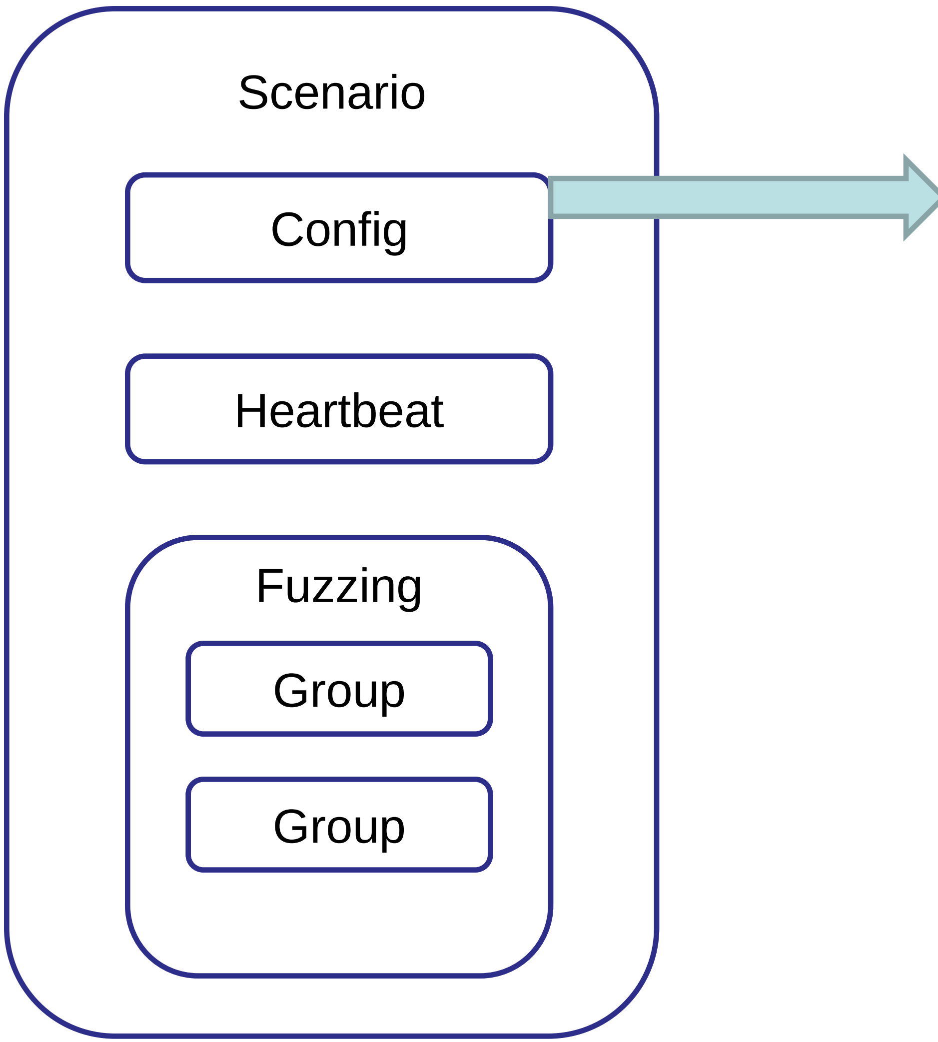
Statefulness

- Sometimes protocols requiring keeping state
- A particular problem for generative fuzzers (mutative fuzzers can usually playback their inputs)
- Need to incorporate responses from target into future fuzz cases
- Examples
  - Fuzzing an FTP server's command line parsing
  - Fuzzing a TCP implementation (sequence and acknowledgement numbers)

Grape

- Generative Fuzzer
- Handles responses for stateful fuzzing
- Rules for generation written in a YAML-like dialect
- Compose rules into fuzz scenarios with Scapy-like syntax
- Pluggable backends – output can be to file, network, etc
- Sensible default low-level protocols – fuzz HTTP without fuzzing (or thinking about) IPv4
- Heartbeat-based monitoring
- No crash data collection yet

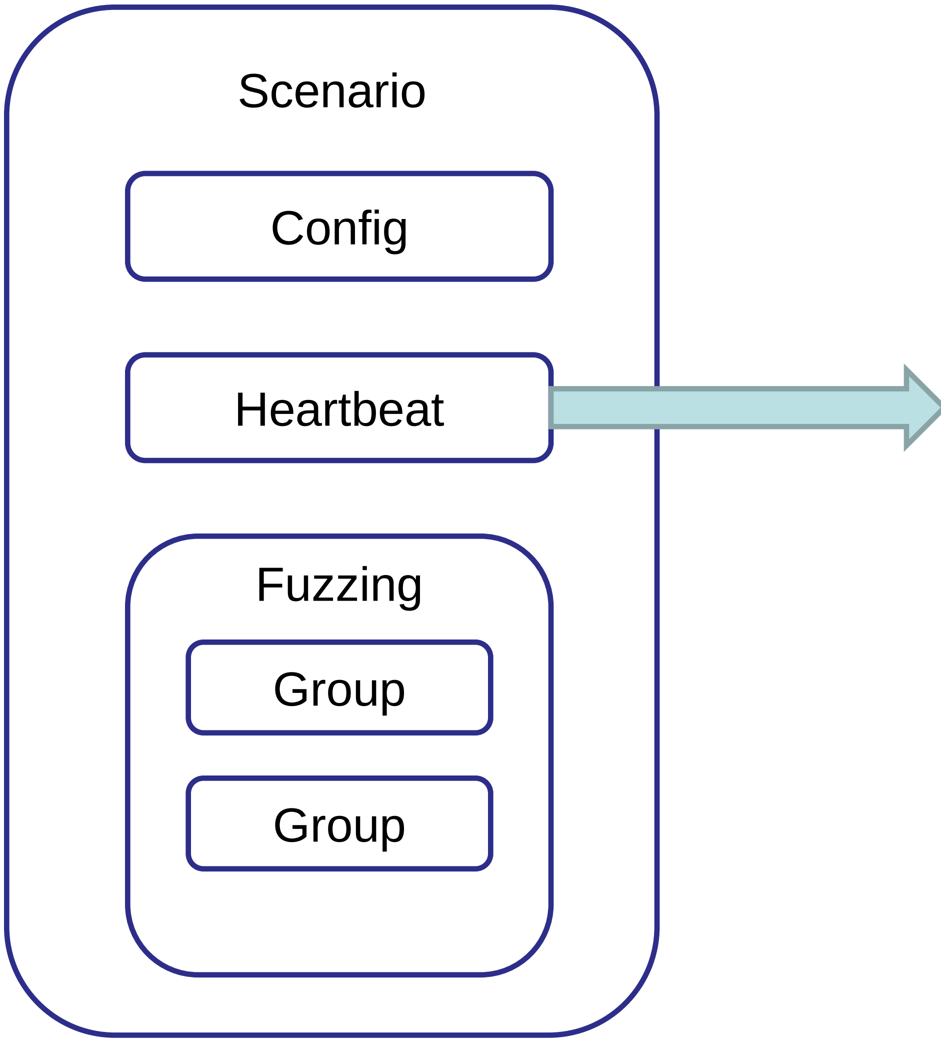

Scenario
Config
Heartbeat
Fuzzing
Group
Group



Configuration for this scenario.
E.g. setting a path and low layer.

Example:

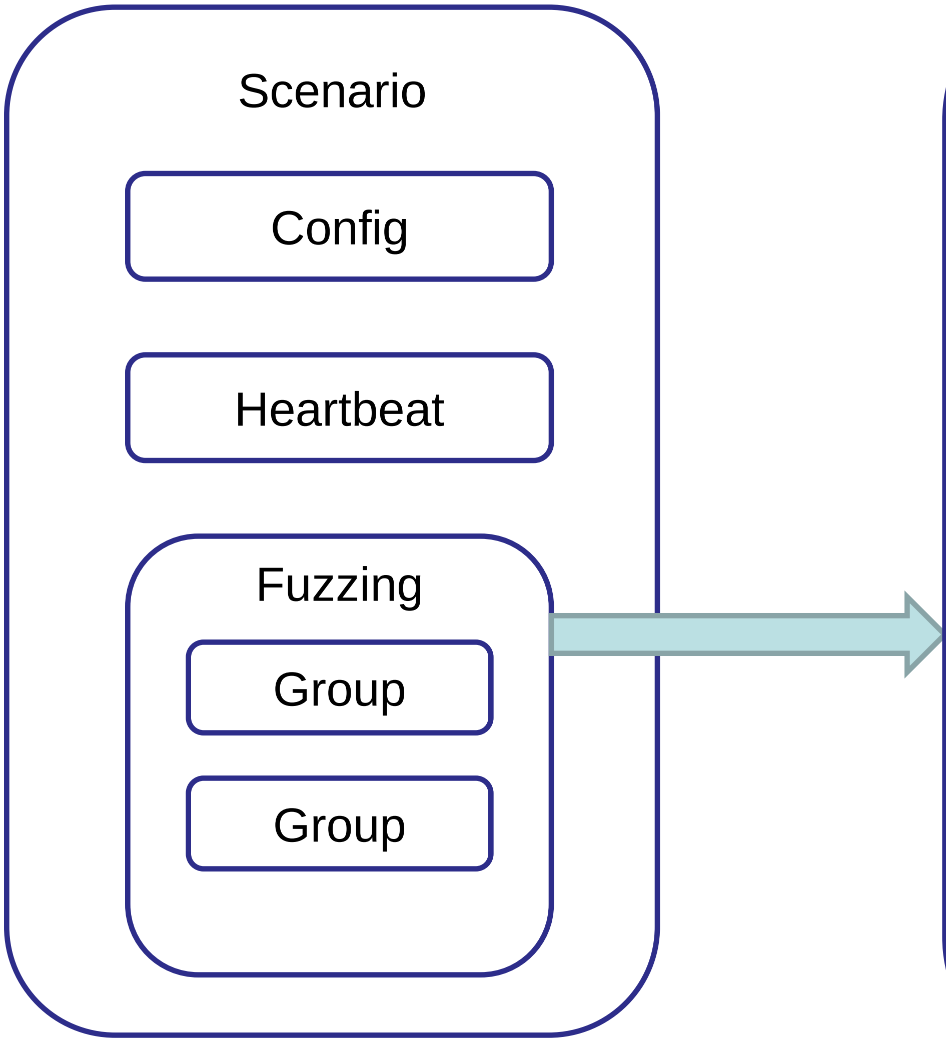
```
config {
  rule_path: http/
  low_layer: ether/ipv4/tcp
}
```



Monitor the target with a "heartbeat"

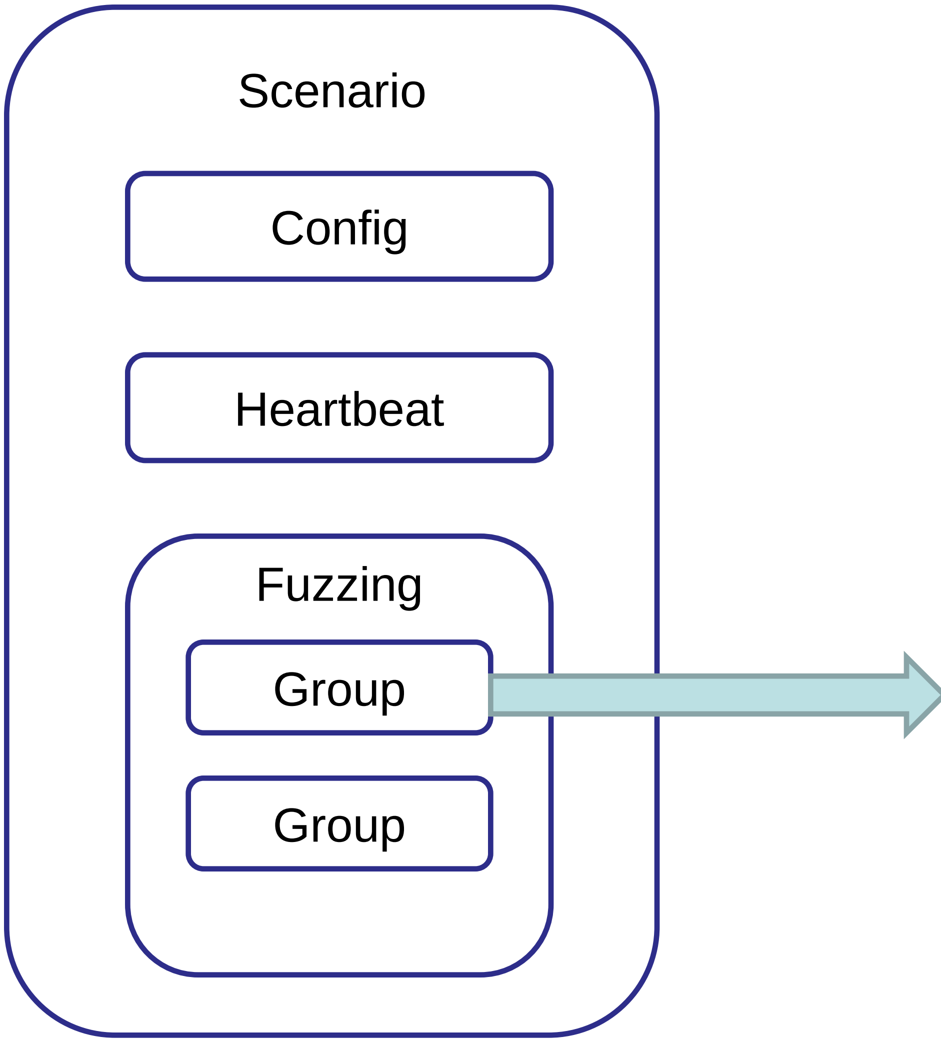
Example:

```
heart_beat {
  group {
    send: http_head
    recv: recv_http_head
  }
}
```



The "fuzzing" category describes a set of "groups" which control the real fuzzing

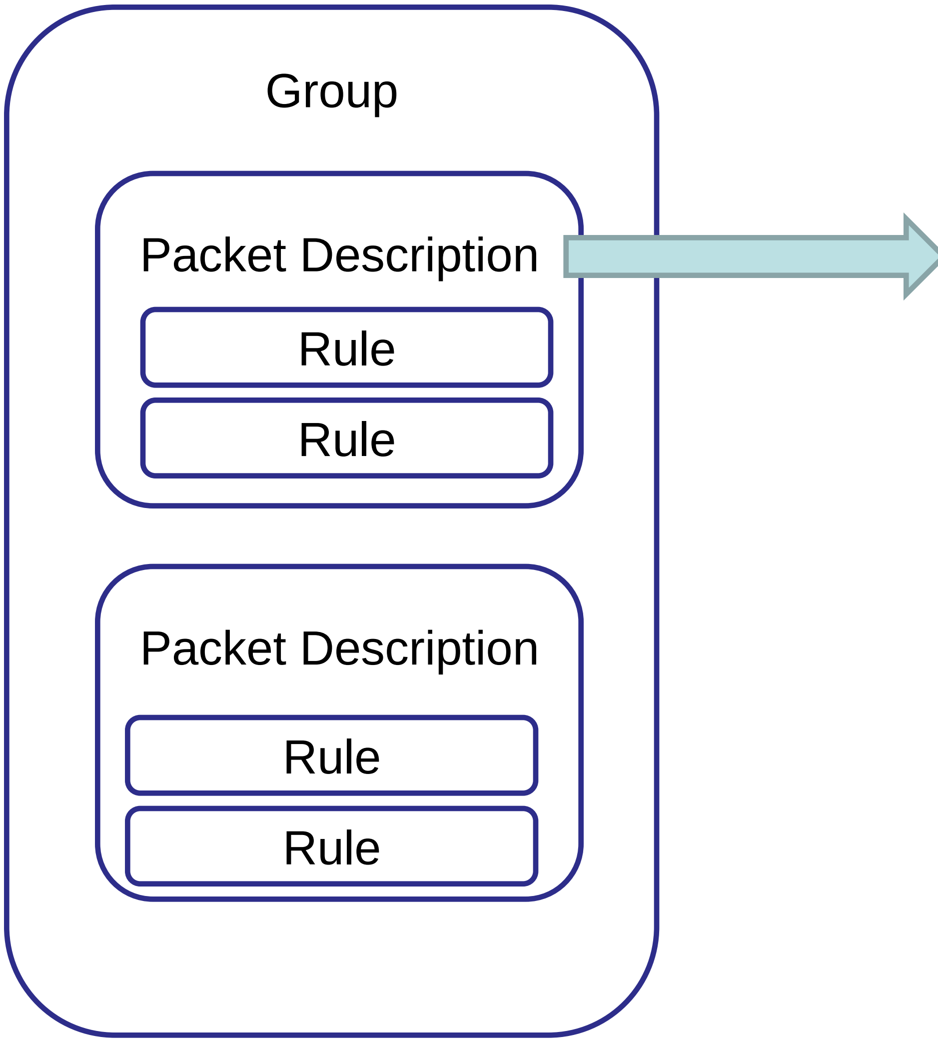
```
fuzzing {
 group {
    …
  }
  group {
    …
  }
}
```



A group describes a "conversation" or series of generations.

Example:

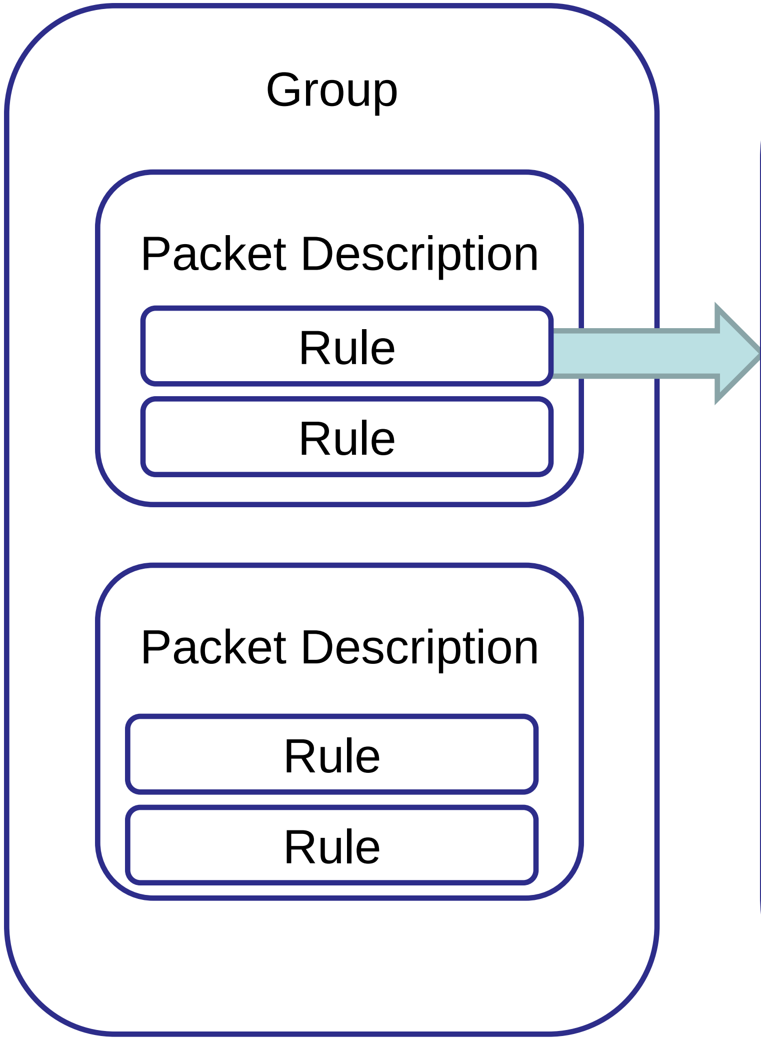
```
group {
  send: http_init
  recv: recv_basic
  send: tricky_stuff
}
```



A packet description is one “send” or “receive” line from the groups:

Example:

send: http_init



A rule describes how an actual input or series of inputs is created (this is the generative bit!):

Example:

```
mostly_harmless:
    method/s: ["GET", "PUT"]
    space/s: " "
    path/s: ["/index.html", "/“]
    http/s: " HTTP/1.1¥r¥n"
    done/s: "¥r¥n"
```

Simple Interactions

send: send this to (network/ a file)

recv: Receive this response (network only for now)

recv rules match the incoming data with certain rules

–If no match, skips to next fuzzing fuzzing case

Note: no ‘real’ flow control

–Use several groups, instead

Packet Structure Description

Here's where we took inspiration from Scapy

There's 'layers'

```
ether/ipv4/tcp(syn:1)/payload(data:"AAAAA")
```

'/' separates layers, parentheses allow overwriting of named values inside the 'rules'

'sublayers' can be placed in parentheses

```
ether/ipv6(routing(type:0))/udp/random(50)
```

Rule Definitions

The structures of generated inputs are composed from 'rules' These rules are defined in separate files.

YAML-inspired syntax, but not really YAML

A Rule:

```
gif_basic:
    signature/s3                  : "GIF"
    version/s3                    : ["89a","87a"]
    logical_screen_width/I2       : 32
    logical_screen_height/I2      : 52
    global_color_table_flag/b1    : 1
    color_resolution/b3           : 7
```

Primitive Definitions II

```
version/s3: ["89a","87a"]
```

Primitives are given by name, followed with a type and a length, and then possible values for that primitive to take.

These values are automatically used in fuzzing.

Type is one of:

- I: Big Endian Integer (that's a capital i)
- i: Little Endian Integer
- S: Symbol
- s: String
- B: Binary
- b: Bitfield

Lengths are in bytes, except for bitfields, where they are in bits.

Rule Definitions

The structures of generated inputs are composed from ‘rules’ These rules are defined in separate files.

YAML-inspired syntax, but not really YAML

A Rule:

```
gif_basic:
    signature/s3                : "GIF"
    version/s3                  : ["89a","87a"]
    logical_screen_width/I2     : 32
    logical_screen_height/I2    : 52
    global_color_table_flag/b1  : 1
    color_resolution/b3         : 7
```

Fuzzing Combinations

Fields like `version/s3: ["89a","87a"]` with multiple values are automatically fuzzed by the fuzzing engine.

Output is generated such that every value given for a field is present at least once in the output. One field per output is 'fuzzed'; that field is iterated over. All others take their leftmost value.

Fuzzing is not combinatorial, however:

```
version/s3: [“A",“B"]
width/I2: [1, 2, 3]
```

produces 4 combinations:

```
(“A”, 1) (“B”,1) (“A”, 2) (“A”, 3)
```

Combinatorial fuzzing

- We can also have fields that we want to fuzz as a “combination”. i.e. This Rule:

```
CombinationMultiFieldFuzz:
    value1%combo1/s1: [“A” , “B”]
    value2%combo1/I1: [1, 2, 3]
```

Produces the following 6 combinations:

```
(“A”, 1) (“B”,1) (“A”, 2) (“B”, 2”) (“A”, 3) (“B”, 3)
```

Response Definitions

Responses are matched against response rules. These are similar to the generation rules. Specifying a value indicates that part of the response should match that value.

_ is “Don’t care”, and matches anything

Can also capture values using $() syntax:

```
recv_tcp:
     src/i2  : _
     dest/i2 : _
     seqno/I4: $(sequence_number)
```

Captured values are available as variables.

Response Definitions II - Regexes

Response Definitions can include simplified regexps for string matching

```
HTTP:
  response: [“%s %d %s¥r¥n”,  $(version), $(code),
                                $(status)”]
```

These are powered by Oniguruma; the results of the scanf –style capture directives get saved to corresponding variables.’

Real regexes can also be used for more power (i.e. non-scanf-style).

Response Definitions III

Primitives in responses can be marked with an asterisk ‘*’ to indicate 0 or more of the primitive should be matched.

Useful for matching higher-level patterns:

HTTP:

```
    header-line/s*: “%s¥r¥n”
    cache-expires/s: “EXPIRES blah blah ¥r¥n”
    header-line/s*: “%s¥r¥n”
    done/s: “¥r¥n”
```

This matches an expires line at any point in the header

Variables

```
tcp:
        srcport/I2: 0
        dstport/I2: $tcp_dst_port
        …
```

The syntax '$tcp_dst_port ' inserts the value of a variable named 'tcp_dst_port'.

Variables can be set by the user initially, captured from incoming packets, or calculated by macro statements.

Symbols

- Symbols go inside <angle_brackets>
- Similar to variables but for internal use within the rules.
- Get substituted like rules

```
fuzzable_thing:
  type%comb/i1     : [0,1,2,3, 256]
  length%comb/i1   : [1,127, 128, 255, 256, 32767, 32768, 65535, 65536]
  data%/comb/B     : $repeat(<padding>, ($ivalue(<length>) + 1) / 2)

padding:
  data1%comb/I1 : [0,1,2,3,4]
  data2%comb/I1 : [0,1,2,3,4]
```

Macros

```
ipv4[6]:
  version/b4        : 4
  header_length/b4 : [($ilength(<ipv4>) -$ilength(<payload>)) / 4, 0, 15]
  dscp_or_tos/S     : [ <tos>, <dscp> ]
  packet_length/I2 : [$ilength(<ipv4>), 1, 16, 32]
```

Various macros are provided, e.g. `$ilength(<symbol>)`

Arithmetic permitted – header_length can be the length of the whole ipv4 block, minus the length of the payload block, divided by 4.

Other macros include `$tcp_checksum`, `$md5`, `$repeatA`

Macro Example

```
void Macros::macro_irandom(Field *f, Var &out,
                           int argc, Var *argv)
{
      int ret = rand();
      out.setInteger(ret);
}
```

The macro interface is still work-in-progress.

`Var` types hold values used during generation; the result of a macro can be set by calling '`setInteger`', '`setString`', etc, on the 'out' argument of the macro.

`argv` is an array of `argc` pointers to `Vars`.

Backends

How the generated data is actually used.

Currently Provided:

Raw-Ethernet IPv4 IPv6 UDP TCP HTTP File

Lower level network backends use raw sockets and libpcap

Higher level network backends use OS provided sockets

Multiple Backends

Backends can be named

command: ether/ipv4/tcp

data: ether/ipv4/tcp(tcp_dst_port: 20)

Packets can be sent to any named backend

send(command): ftp/login(uname: ”foo”, pword: “bar”)

send(data): ftp/payload(data: “hogehogehoge”)

Sent to default(first) backend if no name specified.

Connection-based backends automatically opened on first send

Monitoring

- Currently an optional 'heartbeat' can be defined
- Detects when the target stops responding
- Usually, ICMP or ICMPv6 Echo Requests (pings)
- Can specify heartbeat interval (once every n packets)

HeartBeat

```
heart_beat {
  group {
    send: ipv4/icmp(icmp_echo_req)
    recv: recv_ipv4/recv_icmp(recv_icmp_echo)
  }
}
```

The monitoring heartbeat is specified like any other fuzzing rule.

Heartbeat can have a different backend.

Example – IPv6 Fuzzing

```
ipv6:
    version/b4         : 6
    trafficclass/b8    : 0
    flowlabel/b20      : 0
    packet_length/I2: $ilength(<payload>) + $ilength(<headers>)
    next_header/I1     : $id(<next>)
    hoplimit/I1        : [127, 255, 0]
    src_address/B16    : $ipv6_addr($ipv6_src)
    dest_address/B16   : $ipv6_addr($ipv6_dst)
    headers/S          : <next_sublayer>
    payload/S          : <next_layer>
```

Generates an IPv6 header, and continues into the extension headers.

IPv6 Fuzzing - contd

For IPv6, instead of fuzzing values we fuzz structures -

Various combinations of chains of extension headers and associated options:

```
send: ipv6(hopbyhop(home_address/quick_start)/routing/esp)/tcp
send: ipv6(hopbyhop(home_address/endpoint_ID)/routing/esp)/tcp
send: ipv6(hopbyhop(home_address/tunnel_limit)/routing/esp)/tcp
send: ipv6(hopbyhop(home_address/router_alert)/routing/esp)/tcp
```

Of course, these are generated by a script.

Example: TCP Fuzzing

```
tcp[0x6]:
        srcport/I2: 0
        dest/I2: $tcp_dst_port
        seqno/I4: 0
        ackno/I4: 0
        dataoff/b4: ($ilength($<opts>) / 4) + 5
        reserved/b4: 0
        cwr/b1: 0
        ece/b1: 0
        urg/b1: 0
        ack/b1: 0
… etc.
```

TCP control bits can be set using overwrites in the scenario file.

TCP Scenario Excerpt

```
group{
send: tcp(seqno:1747422, srcport: 6295, syn: 1, cwr:1, ece:1)
recv: recv_tcp
send: tcp(seqno:$recv_ack, srcport: 6295, ack: 1, ackno: $recv_seq + 1 ,
cwr:1, ece:1)
recv: recv_tcp
send: tcp(seqno:1747423, srcport: 6295,cwr:1, ece:1)/tcp_payload
recv: recv_tcp
send: tcp(seqno:1747449, srcport: 6295,fin: 1, cwr:1, ece:1)/tcp_payload
}
```

Scenario file uses overwrites to control the higher-level behaviour to comply with the TCP protocol.

Demo One – IPv6

- Quick ipv6 fuzzing demo against Windows 7

Demo Two: Router(a)

- DoS

Limitations

- Speed
  - Research quality code
- Expressiveness
  - Flow Control in scenarios

Small set of backends at present

Future Work

- Speediness
- Flow Control
- More Backends
- Macro programming for everyone
  - Scripting language
- More sophisticated monitoring
  - Likely requires cooperation with vendors for embedded devices
  - Develop a protocol?
- More file-oriented fuzzing support (spawning processes to open generated files, etc)

Thank you!

Questions?

Fourteenforty Research Institute, Inc.
http://www.fourteenforty.jp

Nicholas Green
green@fourteenforty.jp