The ONL Plugin Environment (Jan 2005)

Ken Wong
Washington University

kenw@wustl.edu
www.arl.wustl.edu/~kenw

Topics

- The plugin environment (Overview)
- Plugin examples
- Plugin management
- Inside plugin management
- Inside the plugin environment
- Steps in writing a basic plugin
- A tour thru the COUNTER plugin
- Exercises
The Plugin Framework

- A plugin can extend the capabilities of a PP
  - Examine or modify packet headers and/or bodies
  - Delay packets
  - Produce additional packets
  - Modify packet shims

- Plugins are installed along a packet’s data path

- The plugin framework follows an OO paradigm
  - A plugin instance (object) has its own local variables (state) and is created from a plugin class
  - Filters can direct their matching packets to plugin instances (NOT plugin classes)
  - The code for a plugin implements the plugin abstraction (e.g., load, create, bind, handle pkt, handle msg)
The Plugin Abstraction

- User commands
  - **Load/Unload** a plugin (class) into/from a PP
  - **Create/Free** an instance of the plugin (class)
  - **Bind/Unbind** a plugin instance to/from a filter
  - **Send a message** to a plugin instance

- Existing code handles user commands in a basic way

- A filter-plugin binding causes matching pkts to be sent to the plugin instance for processing
  - The user writes a `<Plugin>_handle_packet` function to handle packets passed to it by the FPX

- The abstraction is implemented by the plugin code in cooperation with the SPC kernel code

---

Plugin Control Path

- **CP** → **PP**
- **PP** → **ATM Switch Core**
- **external links**
- **CP** → **PP** → **SPC** → **FPX**

- **Control Cells**
  - **Load plugin**
  - **Instantiate plugin**
  - **Install filter**
  - **Reverse ops**
  - **Bind plugin+filter**
  - **Management**
Existing Plugins

- Build custom plugin by modifying existing plugin
- Plugin source code
  - `$WUARL/msr/rp/plugins/
  - e.g., WUARL = ~onl/repository/wu_arl
- Existing Plugins
  - template: Plugin framework
  - COUNTER: Counts pkts
  - delay: Delays pkts
  - testEM: Example of adding/deleting EM filters
  - syn_demo: SYN attack mitigation

Ex. 1 (COUNTER Plugin + EM Filter)

- EM filter for ping traffic thru COUNTER plugin
  - 192.168.1.112, 0, 192.168.1.48, 0, 1 (ICMP), fwd port=2
- Features
  - Simple plugin installation and management
COUNTER Demo Running

ping -c 10 -s 1000 192.168.1.48

BW about 1.1 x 8 Kbps

ping 10 1000-byte pkts at 192.168.1.64 (port 3), one pkt per sec

plugin binding = qid 8

protocol 1 ➔ icmp

Ex. 2 (Delay Plugin, GM Filter)

CP plugin mngt and messages

iperf TG 192.168.1.112

Delay(20 ms) Plugin

Ingress GM filter

Plugin responds to the following msgs:
Get queue dump and statistics
Set delay
Get statistics (#pkts, #dropped, #fwd)

GM filter for iperf traffic thru delay plugin
0.0.0.0/0, 0, 192.168.1.64/32, 0, TCP, fwd port=3

Features
More complicated plugin processing/control
Delay Demo Running (1)

iperf sees 23.8 Mbps

BW about 64 KB/20 ms = 25.6 Mbps

Ex. 3 (SYN Flood Mitigation)

- Additional Features
  - TCP connection monitoring
  - Triggered generation of TCP RST (Reset) pkt to terminate incomplete TCP connection
  - Shim modification
  - Installation of EM filter
  - On/Off control of plugin

- SYN Flood Attacker
  - Sends many TCP SYN pkts to target
    - Signals new TCP connection
  - But never responds to server's SYN-ACK pkt
  - Consumes memory which records new connections
SYN Flood Demo Plugin (1)

Egress

GM1: da=192.168.1.64/32, dp=80, tcp

Ingress

EM: 192.168.1.48, 28037, 192.168.1.64, 80, tcp

GM2: sa=192.168.1.64/32, dp=80, tcp

SYN Demo Components

Web Client
Periodic image get

Attack Controller
On/Off

Plugin Controller
On/Off

Browser
192.168.1.48
ssh tunnel

Flood server with TCP SYN packets
**Web Client Interface**

- Start/Stop image transfer
- Delay: Interimage period
- URL: Image directory (Note '/')
- Image Activity Log

- Initialize/Connect/Disconnect
- Start/Stop image transfer
- Rate: SYN pkts per sec
- Attack Daemon socket addr
- Spoofed sender address
- Web server socket addr

- Initialize/Connect/Disconnect
- Start/Stop (Enable/Disable)
- Repeat Switch
- Plugin Mgr socket address
- Activity period
- Plugin Location
  - (port, instance)

**SYN Demo Running**

- Plugin ON
- Plugin ON
Plugin Control Path

A Plugin Instance (Object)

- "Looks" like an OO object
  - Derived from a plugin definition (like an OO class)
  - Has local state (instance variables) and global state (variables)
- Written in C (NOT C++)
- Instance must provide these services
  - "Handle" a packet
  - Handle control messages
- Plugin environment (in kernel) with assist from class provides these services
  - Loading/Unloading class
  - Binding/Unbinding class
  - Creating/Deleting instance
Plugin Management

- Suppose we have an SPC plugin object module ("combined.o")

**Install Phase**
- Load the plugin
- Create plugin instance I
- Add a filter F
- Bind instance I to filter F

**Deinstall Phase**
- Unbind instance I from filter F
- Remove a filter F
- Delete plugin instance I
- Unload the plugin

**Plugin control messages**
- Plugin management should be in RLI BUT ...
  - For now, use command interface or
  - Use PM_demo (curses interface) [ NOT official ]

Plugin Management Commands

- **pluginDownload**
  - Load a kernel object module to an SPC (at a port)

- **sendcmd**
  - Send a control message to an SPC module
    - e.g., PCU (Plugin Control Unit), DEBUG (Debugging), DQ (Distributed Queueing)
  - Send a control message to a plugin instance

- **cfy**
  - Send a control message to the classifier
    - Bind/Unbind filter to/from plugin instance
    - Control delegated from CP to SPC
The SPC Control Path

- 'sendcmd -t 100000 -w info -p 6 -c rp_pcu -s create -i 432'
  - Send command to **PCU** (Plugin Control Unit) at port 6
  - **Subcommand**: Create an instance of plugin class 432

sendcmd

**Command Protocol**
- **Command Cycle**: CP sends command message to a port and expects a reply message indicating success or failure
- **Command Transaction**: One or more command cycles terminated by an EOF reply message

**Some Top-Level Commands**
- `set_debug/get_debug`: Set/Get debug flags and mask
- `port_init`: Set port number; enable DQ
- `rp_pcu`: Send msg to plugin Control Unit
- `rp_inst`: Send msg to plugin instance
- `policy`: Manage policy object
### Preliminaries

**Plugin management commands**
- Normally in a CVS tree but we'll use `/users/oni/bin/*`
- Approach 1: Set up aliases to commands
  - source `/users/oni/export=/srcthis.csh` or `=srcthis.bash`
- Approach 2: Manual
  - Add `/users/oni/bin` to your PATH environment variable
    - csh: `set path=$(path /users/oni/bin)`
    - bash: `export PATH="$PATH:/users/oni/bin"`
  - setenv SND "'/users/oni/bin/sendcmd -t 100000 -w info""

**Help**
- `sendcmd -h`
- `cfy -h`
- `/users/oni/bin/usage-*.txt` are help files for some common commands
**Plugin Management Messages (1)**

- **Load plugin module** $M$ **which has entry point** $E$
  
  » pluginDownload -d -e $E -s $M -p $P
  
  » e.g., pluginDownload -d -e COUNTER -s combined.o -p 6

- **Create instance of plugin class id** $C$
  
  » $SND -c rp_pcu -s create -p $P -i $C
  
  » e.g., $SND -c rp_pcu -s create -p 6 -i 432

- **Bind instance** $I$ **to filter queue** $Q$
  
  » cfy -p $P -i $I -q $Q -- bind
  
  » e.g., cfy -p 6 -i 0 -q 8 -- bind

- **Send data to instance** $I$
  
  » $SND -p $P -c rp_inst -d $I -d <Datum> -d <Datum> ...
  
  » e.g., $SND -p 6 -c rp_inst -d 0 -d 0

---

**Plugin Management Messages (2)**

- **Unbind instance** $I$ **from filter queue** $Q$
  
  » cfy -p $P -i $I -q $Q -- unbind
  
  » e.g., cfy -p 6 -i 0 -q 8 -- unbind

- **Delete instance** $I$
  
  » $SND -c rp_pcu -s free -p $P -i $I
  
  » e.g., $SND -c rp_pcu -s free -p 6 -i 0

- **Unload (Remove) plugin at location** $L$
  
  » $SND -c rp_pcu -s unload -d $L -p $P
  
  » e.g., $SND -c rp_pcu -s unload -d 0 -p 6
### Example

```bash
### Suppose filter directing traffic to SPC has been installed at port 6
$SND -c get_debug -p 6  # get debug status on port 6
$SND -c set_debug -l error -m all -p 6  # output just errors, not all msgs
$SND -c policy -s set_dflags -d 2 -p 6  # where to send debug msgs:
    # 1:local, 2:remote, 3:both
pluginDownload -d -e COUNTER -s combined.o -p 6  # download plugin
$SND -c rp_pcu -s create -p 6 -i 432  # create instance of plugin id 432
cfy -p 6 -i 0 -q 8 -- bind  # bind inst 0, qid 8
    ### before sending 10 ping pkts
$SND -c rp_inst -p 6 -d 0 -d 0  # snd msg to instance 0, command 0
    ### repeat after sending 10 ping pkts
$SND -c rp_inst -p 6 -d 0 -d 0  # snd msg to instance 0, command 0
    ### send 1 pkt here
$SND -c set_debug -l verbose -m all -p 6  # VERBOSE output for all modules
cfy -p 6 -i 0 -q 8 -- unbind  # unbind inst 0, qid 8
$SND -c rp_pcu -s free -p 6 -i 0  # free inst 0
$SND -c rp_pcu -s unload -d 0 -p 6  # class
    ### extra stuff
$SND -c rp_pcu -s clist -p 6  # monmsgs: class list
$SND -c rp_pcu -s ilist -p 6  # monmsgs: instance list
```

---

### PM_Demo Tool (1)

- **PM_Demo uses curses**  *** Not Official ***
  - ```bash
    cd <PluginRootDir>; PM_demo -P .
  ```
  - **Expects ".plugins" plugin table**
  - **CAVEAT: Need atleast 32 x 96 xterm or cygwin window**
```bash
```
```bash

---

28. Ken Wong, 3/14/2005

Washington University in St. Louis
PM_Demo Tool (2)

- '>>>' cursor in Object frame ➔ Current object
- Up/Down arrow moves cursor
- 'q' command exits

Pause For Plugin Management Demo

- Plugin management commands
- PM_demo tool
  - Without monmsgs
  - With monmsgs
**Plugin Management Demo (1)**

- Load plugin:
  - `load plugin
  - successful load
- Create instance:
  - `create instance
- Bind:
  - `bind
- Send msg to plugin:
  - `snd msg to plugin
- Repeat after 10 pkts:
  - `10 pkts
- Unload:
  - `unload

**Plugin Management Demo (2)**

- Current port:
  - `6 # current port
- No noise while loading:
  - `n 0 # NO noise while loading
- Load COUNTER plugin:
  - `l 432 # load COUNTER plugin
- Create instance:
  - `c 432 # create instance
- Bind to filter queue 8:
  - `b 8 0 # bind to filter queue 8
**Plugin Debugging Output**

- **Select debug output type(s) and destination**
  - Output (Verbose) type(s)/level(s)
    - verbose, info, trace, warn, error, critical
  - Can have more than one (-l info -l warn) but usually pick one
  - **Destination parameter**
    - 1: local (console if it exists); 2: CP; 3: Both

- **Produce output thru MSR_Debug macro in code**
  - MSR_DEBUG((MSR_DEBUG_PLUGIN | MSR_DEBUG_LEVEL_INFO, "ID = %d got pkt %d\n", id, pkt_count));

- **Capture debug msgs at CP using 'monmsgs' command**
- *** NOTE *** Doesn't take much to swamp out SPC

---

**Debug Output Example 1**

- **COUNTER-debug.c code**
  - MSR_DEBUG( ) macro sends output to CP
    - MSR_DEBUG_PLUGIN: Category (i.e., source of output)
    - MSR_DEBUG_LEVEL_INFO: Output type/level
    - Formatting is like printf

```c
void COUNTER_handle_packet(struct rp_instance *this, void *pkt) {
    struct COUNTER_instance *inst = (struct COUNTER_instance *)this;
    inst->pkt_count++;
    MSR_DEBUG((MSR_DEBUG_PLUGIN | MSR_DEBUG_LEVEL_INFO, 
                 "COUNTER_handle_packet: Instance ID = %d got pkt %d\n", 
                 this->instanceid, inst->pkt_count));
}
```

---
monmsgs Output

- Must put SPC in verbose mode
  - `$SND -c set_debug -l verbose -m all -p 6`

Plugin Development

- Is like writing code for a remote kernel
  - There is no directly connected console
    - There is monmsgs at the CP
  - There is no memory protection ➔ Easy to crash kernel
  - Can require knowledge of detailed pkt representation
  - There is no easy access to a debugger
    - Like flying almost blind
  - The development cycle is very long

- Advice
  - Test incrementally
  - During testing, use debug msgs efficiently
Steps in Writing a Basic Plugin

- Choose a plugin name
- Choose a unique non-negative integer for a plugin id
- Define the *extended instance variables*
  - These extend the variables in the base class
- Modify the std plugin initialization code
- Modify the std packet processing code
- Modify the std plugin control msg handling code
- Compile the plugin code into the *NetBSD* object module 'combined.o'
  - The code includes standard routines that assist during plugin loading/unloading, creating/freeing, binding/unbinding

The COUNTER Plugin (1)

- Function: Count packets
- Plugin Directory/Name: COUNTER
- Entry Function Name: COUNTER
  - Kernel entry point for loadable kernel module
- Plugin Id: 432
  - Must be unique within a port
- Extended instance variables
  - By convention, form is defined in COUNTER.h header file
  - Allocation is done in COUNTER_create_instance()

```c
struct COUNTER_instance {
    struct rp_instance base;  // base part
    int     pkt_count;        // pkt counter
};
```

Only used in PM_demo tool
A Plugin Instance (Object)

- "Looks" like an OO object
  - Derived from a plugin definition (like an OO class)
  - Has local state (instance variables) and global state (variables)
- Written in C (NOT C++)
- Must provide these services
  - "Handle" a packet
  - Bind/Unbind to/from a filter
  - Handle control messages

COUNTER.h (1)

- Plugin id used in loading plugin/creating instance
- State variables of each instance
  - Every plugin instance has the same base class variable
  - Extend base with the class-specific data

```c
#define COUNTER_ID 432;
struct COUNTER_instance {    // COUNTER instance
  struct rp_instance root_instance;    // base class part
  int pkt_count;    // instance specific data
};

load plugin, create instance

void COUNTER_init_class();    // function prototypes
struct rp_class *COUNTER_get_class();
struct rp_instance *COUNTER_create_instance
  (struct rp_class *, u_int32_t);
```
Functions

- All functions except pkt handling code
  - Initiated from the Control Processor (CP) by sending a command to the SPC's Plugin Control Unit (PCU)
  - PCU forwards msgs to plugin instance

- Management functions
  - `COUNTER_{load, create_instance, bind_instance, handle_packet, handle_msg, unbind_instance, free_instance, unload}`
  - In most cases, only need to modify handle_packet, handle_msg, create_instance and free_instance

- Other functions
  - `COUNTER_init_class`: Initialize class variables (e.g., id)
  - `COUNTER_get_class`: Return ptr to class structure
  - `COUNTER`: Kernel entry point (inserts load/unload ptrs into loadable kernel module table)

---

COUNTER_create_instance (1)

- Allocate memory for object (instance) state
  - Table of standard methods (e.g., handle_packet, handle_msg)
  - Class-specific data member 'pkt_count'

- Initialize instance's method table and state variables
  - e.g., `myinst->rootinstance.handle_packet = COUNTER_handle_packet`
  - `my_inst->pkt_count = 0`
COUNTER\_create\_instance (2)

```c
struct rp_instance *
COUNTER\_create\_instance(struct rp_class *myclass,
   u_int32_t instanceid) {
  struct COUNTER\_instance *myinst;
  MSR\_PLUGIN\_MALLOC(myinst, struct COUNTER\_instance *,
    sizeof(struct COUNTER\_instance),
    M\_MSR, M\_WAITOK);
  if (myinst == NULL) return NULL;
  myinst->rootinstance.rpclass = &COUNTER\_class;
  ... .handle\_packet = COUNTER\_handle\_packet;
  ... .free\_instance = COUNTER\_free\_instance;
  ... .bind\_instance = COUNTER\_bind\_instance;
  ... .unbind\_instance = COUNTER\_unbind\_instance;
  ... .handle\_msg = COUNTER\_handle\_msg;
  ... .instanceid = instanceid;
  myinst->pkt\_count = 0;
  return (struct rp_instance *)myinst;
}
```

COUNTER\_handle\_packet

- Processes pkt
  - Increment the counter 'pkt\_count' in instance structure
  - Packet will be forwarded
- Note division of concerns
  - 'struct COUNTER\_instance' contains object state of interest to both PCU and plugin instance
  - But plugin instance only maintains its part: 'pkt\_count'

```c
void
COUNTER\_handle\_packet (struct rp_instance *this,
   void *plist) {
  struct COUNTER\_instance *inst =
    (struct COUNTER\_instance *) this;
  inst->pkt\_count++;
  return;
}
```
COUNTER_handle_msg (1)

- Responds to CP messages
  - CP pulls data from SPC by sending msg to an instance
  - Msg handled by COUNTER_handle_msg()

- Standard command protocol
  - 1 AAL0 cell to transmit a command to SPC
  - Expects a reply stored in 1 AAL0 cell

- ATM cell payload is 48 bytes

<table>
<thead>
<tr>
<th>ver</th>
<th>len</th>
<th>cmd/status</th>
<th>seqno</th>
<th>flags</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>instance id</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

  cmd = MSR_CTL_RP_INST
  Not used in COUNTER

  11 words of user data

COUNTER_handle_msg (2)

- Responds to msg from CP asking for pkt count
  - Puts pkt count into buffer
  - Returns msg length in *len argument
  - Returns status of 0

  int COUNTER_handle_msg (struct rp_instance *this, void *buf,
                           void *flags, u_int8_t seq,
                           u_int8_t *len) {
    struct COUNTER_instance *inst =
      (struct COUNTER_instance *) this;
    u_int32_t *vals = (u_int32_t *) buf;
    *vals = htonl(inst->pkt_count);  // count
    *len = sizeof(u_int32_t);         // msg size
    return 0;
  }

  points to user data area
  points to 1st user data word
COUNTER_free_instance

- Only necessary action is to free instance memory

```c
void COUNTER_free_instance(struct rp_instance *this) {
    if (this) {
        MSR_DEBUG((MSR_DEBUG_PLUGIN | MSR_DEBUG_LEVEL_INFO,
            "COUNTER_free_instance: Freeing inst %d ((class %d)\n",
            this->instanceid, this->rpclass->classid));
        MSR_PLUGIN_FREE(this, M_MSR);
    } else {
        MSR_DEBUG((MSR_DEBUG_PLUGIN | MSR_DEBUG_LEVEL_WARNING,
            "COUNTER_free_instance: Passed NULL ptr\n"));
    }
}
```

COUNTER_init_class

```c
void COUNTER_init_class() {
    COUNTER_class.classid = COUNTER_ID;
    COUNTER_class.iotype = RP_INTERFACE_TYPE_HLIST;
    COUNTER_class.create_instance = COUNTER_create_instance;
    return;
}
```

- **COUNTER_class.iotype**
  - **RP_INTERFACE_TYPE_PKT**
    - For Read-Only plugins
  - **RP_INTERFACE_TYPE_HLIST**
    - For plugins that want to modify pkt, source pkt, or sink pkt
  - Affects interpretation of ptr argument to COUNTER_handle_packet(this, ptr)
Other Functions

- Handles basic functions specific to application
  - Non-specific work is done by plugin framework of kernel

Functions

- `COUNTER_get_class`: Return ptr to class structure
- `COUNTER_bind_instance`: Zero pkt counter
- `COUNTER_unbind_instance`: Zero pkt counter
- `COUNTER_i`
  - Store load and unload function ptrs into loadable kernel module table using MSR_PLUGIN_DISPATCH macro
- `COUNTER_load`
  - Init class structure and register class with plugin environment
- `COUNTER_unload`
  - Free all plugin instances and deregister class with plugin environment

Compiling the COUNTER Plugin

- `ssh` to NetBSD host (onlbsd1 or onlbsd2)
- `Change directory to plugin source`
- `Edit source code`
- `Enter 'make'`
  - Does a gcc compile producing the object module `combined.o`
Other Plugins

Delay
- Purpose: Delay pkts
- Shows how plugin cooperates with SPC hardclock routine to hold on to pkts for a fixed time length

Syn_demo
- Purpose: Mitigate SYN floods
- Shows plugins can generate pkts (RST) based on a triggering event (incomplete connection time-out) and EM filter installation

Wave video (Still Legacy Code)
- Purpose: Discard high-frequency video frames during congestion
- Shows how application-knowledgeable plugins can proactively use pkt queue lengths to adjust bandwidth usage to avoid pkt drops

Delay Plugin (1)

Delays pkts for D msec
- D defaults to 100 msec and can be dynamically changed
- pdelay_handle_packet( )
  • Queues incoming pkts and
  • Forwards the oldest pkt if it has been delayed enough
- pdelay_callback( ) is called periodically to forward expired pkts

Kernel programming details
- Plugin must interact with SPC’s hardclock routine
- It must insure that use of kernel resources is synchronized
  • Pkt queue is protected from the kernel by disabling interrupts (PLUGIN_SPLCLOCK_FCT) before manipulating the queue and enabling interrupts (PLUGIN_SPLX_FCT) afterwards
**Delay Plugin (2)**

- Uses doubly linked list (tail queue) to hold packets
- Extension Variables

```c
struct pdelay_instance {
    struct rp_instance rootinstance;
    HDRQ_t qhead; // ptr to pkt queue
    int qlen; // # pkts in queue
    int pkt_count; // # pkts seen
    int drop_count; // # pkts dropped
    int fwd_count; // # pkts forwarded
    int max_qlen; // max #pkts queued
    int earliest_depart_time;
    int delay_time; // msec
};
```

**SYN Demo Plugin (1)**

- Uses many functions from delay plugin but demonstrates many new features:
  - TCP connection monitoring
  - Triggered generation of TCP RST (Reset) pkt to terminate incomplete TCP connection
  - Shim modification
  - Installation of EM filter
  - On/Off control of plugin
- Kernel programming details
  - About the same issues as in the delay plugin
  - But much more code and complexity
SYN Demo Plugin (2)

State of incomplete TCP connections is maintained in same tail queue structure used in delay plugin

- But we need a queue for every incomplete connection

```c
struct conn_info {  // descr for each queue
    TAILQ_ENTRY(conn_info) clist;  // fwd/rev ptrs
    u_int32_t  sa, da;  // src-dst IPv4 addr
    u_int32_t  seq;    // TCP seq num
    u_int16_t  sp, dp; // src-dst port
    int        synack_sent; // 1 if SYN-ACK seen
    int        time;     // time of SYN pkt
};
```

For RST packet

```c
struct psyndemo_queue {
    struct conn_info conn_array[CONN_QUEUE_SIZE];  // descr pool
    TAILQ_HEAD(conn_head, conn_info) conn_head, free_head;
};
```

Used to compute expiration time

SYN Demo Plugin (3)

```c
void psyndemo_handle_packet (struct rp_instance *this, void *plst) {
    s = PLUGIN_SPLCLOCK_FCT( ); // Disable interrupts
    if ((hdr = TAILQ_FIRST(hdrs)) == NULL) goto done;
    shim, iph = Addresses of shim and IP header;
    src_addr = htonl(msr_ipsaddr(iph)); // Get IP hdr fields
    dst_addr, {src,dst}_port = Other ip hdr fields;
    curr_time = PLUGIN_CPU_CLOCK_1MSEC_FCT( );
    tcph = TCP hdr ptr;
    switch(SYN, SYN-ACK, ACK, FIN, RST pkts) {  // Act on pkts
        case SYN:    Add item to pending conn queue;
        case SYN-ACK: Update pending conn queue;
        case ACK:    Install EM detour filter;
        case RST | FIN | FIN-ACK: Delete from pending conn queue;
    done:        // Enable interrupts
    return;
}
```
Pause For Demo

- Configure cluster and setup monitoring
- Create plugin instances and bind to GM filters
- Start attack daemon 'synster'
- Start Web server
- Create ssh tunnel for HTTP traffic thru relay node
- Point browser at syndemo directory
  » Configure client, attacker and plugin manager frames
- Displays
  » No attack
  » Periodic plugin operation under continuous attack