A Tutorial on Writing (Binary) Bro Plugins

Robin Sommer

Corelight /

International Computer Science Institute /

Berkeley Lab

robin@corelight.com

robin@icir.org

https://www.icir.org/robin

corelight

INTERNATIONAL COMPUTER SCIENCE INSTITUTE

The Bro Platform

Open Source
BSD License

Analysis

Intrusion Detection | Vulnerabilit. Mgmt | File Analysis | Traffic Measure-ment | Traffic Control | Compliance Monitoring

Platform

Programming Language | Standard Library

Packet Processing

Tap

Network

The Bro Platform

Open Source
BSD License

Analysis

Intrusion Detection | Vulnerabilit. Mgmt | File Analysis | Traffic Measure-ment | Traffic Control | Compliance Monitoring

Platform

Tap

Network

The Bro Platform

Open Source
BSD License

Analysis

Intrusion Detection | Vulnerabilit. Mgmt | File Analysis | Traffic Measure-ment | Traffic Control | Compliance Monitoring

Platform

Log Writers | Built-in Events, Functions, Types | Input Readers

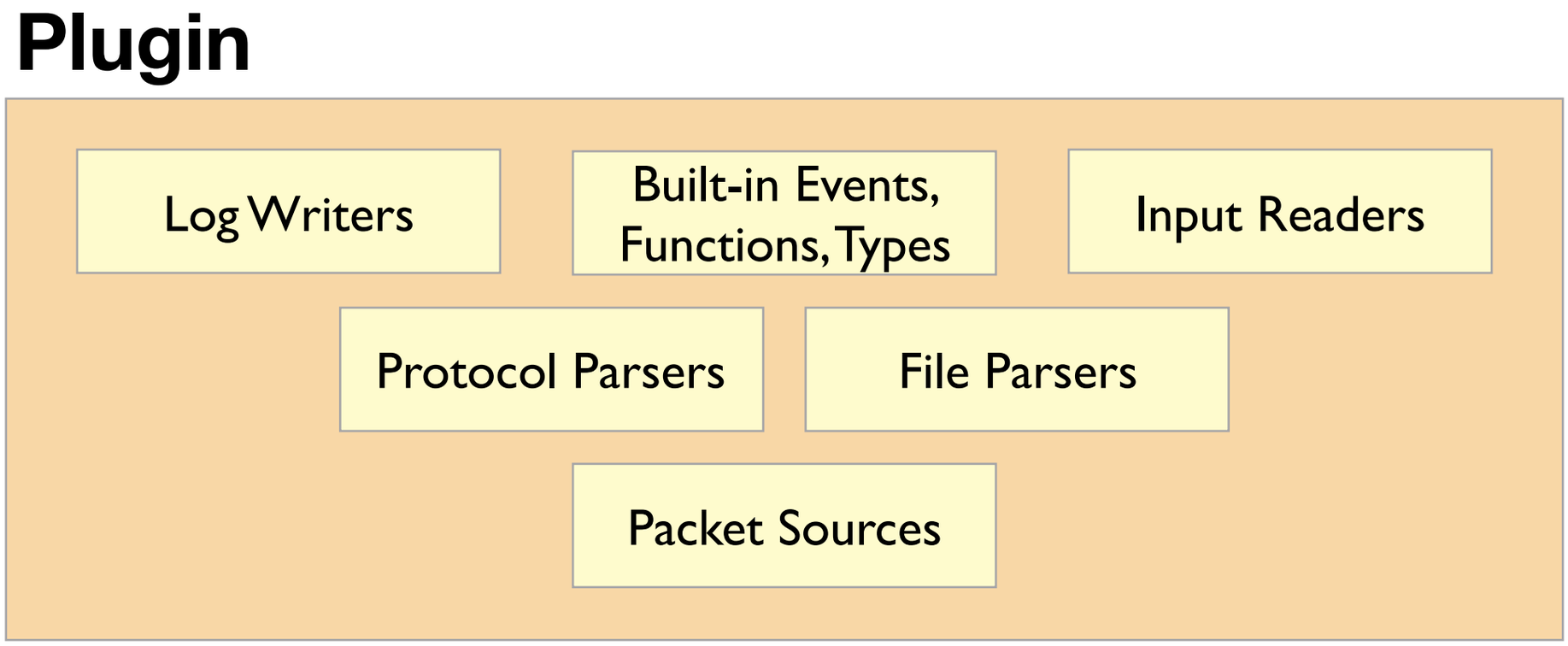
Protocol Parsers | File Parsers

Packet Sources

Tap

Network

Bro Plugins



A *Bro Plugin* is a container for independently compiled *components*, wrapped into a shared library and loaded at startup.

Bro Plugins on GitHub

Kafka

ZeroMQ

PF_RING

AF_Packet

FIX

HTTP/2

Myricom

Elastic

LDAP

Netmap

Community ID

PostgreSQL

Plugin Structure

`<base>/__bro_plugin__`

Marks a directory as containing a Bro plugin, and contains name of plugin

`<base>/lib/<plugin-name>.<os>-<arch>.so`

Shared library implementing plugin and components

`<base>/scripts/`

Bro scripts coming with the plugin, will be added to **`BROPATH`**

Getting Started

Bro comes with a helper script that creates a fully compilable, empty plugin skeleton.

```
# cd src/bro-aux/plugin-support
# ./init-plugin <dir> <namespace> <plugin-name>
```

Getting Started

Bro comes with a helper script that creates a fully compilable, empty plugin skeleton.

```
# cd src/bro-aux/plugin-support
# ./init-plugin <dir> <namespace> <plugin-name>
```

Demo

Component APIs

File Analyzer API

```
class file_analysis::Analyzer {
  virtual void Init();
  virtual void Done();
  virtual bool DeliverChunk(const u_char* data, uint64 len,
                            uint64 offset);
  virtual bool DeliverStream(const u_char* data, uint64 len);
  virtual bool EndOfFile();
  virtual bool Undelivered(uint64 offset, uint64 len);
};
```

```
plugin::component::FILE_ANALYZER
```

Protocol Analyzer API

```
class analyzer::Analyzer {
  virtual void Init();
  virtual void Done();
  virtual void DeliverPacket(int len, const u_char* data,
                             bool orig, uint64 seq,
                             const IP_Hdr* ip, int caplen);
  virtual void DeliverStream(int len, const u_char* data,
                             bool orig);
  virtual void Undelivered(uint64 seq, int len, bool orig);
  virtual void EndOfData(bool is_orig);
  virtual void FlipRoles();
};
```

```
plugin::component::ANALYZER
```

Packet Source API

```
class iosource::PktSrc {
  virtual void Open();
  virtual void Close();
  virtual bool ExtractNextPacket(Packet* pkt);
  virtual void DoneWithPacket();
  virtual bool PrecompileFilter(int index, std::string filter);
  virtual bool SetFilter(int index);
  virtual void Statistics(Stats* stats);

  void Opened(const Properties& props);
};
```

```
plugin::component::PKTSRC
```

Log Writer API

```
class logging::WriterBackend  {
  virtual bool DoInit(const WriterInfo& info, int num_fields,
                      const threading::Field* const* fields)
  virtual bool DoWrite(int num_fields,
                       const threading::Field* const* fields,
                       threading::Value** vals)
  virtual bool DoSetBuf(bool enabled)
  virtual bool DoRotate(const char* rotated_path, double open,
                        double close, bool terminating)
  virtual bool DoFlush(double network_time)
  virtual bool DoFinish(double network_time)
  virtual bool DoHeartbeat(double network_time,
                           double current_time)
};
```

```
plugin::component::WRITER
```

Input Reader API

```
class input::ReaderBackend  {
  virtual bool DoInit(const ReaderInfo& info, int arg_num_fields,
                      const threading::Field* const* fields);
  virtual void DoClose();
  virtual bool DoUpdate();
  virtual bool DoHeartbeat(double network_time,
                           double current_time);

  void SendEvent(const char* name, const int num_vals,
                             threading::Value* *vals);

  void Put(threading::Value** val);
  void Delete(threading::Value** val);
  void Clear();
}
```

plugin::component::READER

Some notes for the advanced plugin writer ….

Plugin Activation

A plugin needs to be *activated* to have an effect.

By default, Bro activates all plugins that it finds in **`BRO_PLUGIN_PATH`** — so nothing to do normally.

But *bare mode* works differently:

- Bro will not activate any plugins by default.
- Bro scripts can activate plugins: **`@load-plugin rsmmr::Demo`**
- Command-line can, too: **`bro -i etc rsmmr::Demo`**
- Environment can, too: **`export BRO_PLUGIN_ACTIVATE=rsmmr::Demo`**

Bro Scripts in Plugins

`<dir>/scripts/`

Will be automatically added to BROPATH

`<dir>/scripts/__load__.bro`

Will be loaded when the plugin gets activated. BiF elements will already be available

`<dir>/scripts/__preload__.bro`

Will be loaded when the plugin gets activated, but before any BiF elements become available

`<dir>/scripts/<ns>/<name>/__load__.bro`

Will be loaded through, e.g., **`@load rsmmr/Demo`**

Hooking into the Script Interpreter

```
class plugin::Plugin {
  virtual int HookLoadFile(const LoadType type, // SCRIPT, SIGNATURES, PLUGIN
                           std::string file, std::string resolved);
  virtual std::pair<bool, Val*> HookCallFunction(const Func* func,
                                 Frame *parent, val_list* args);
  virtual bool HookQueueEvent(Event* event);
  virtual void HookDrainEvents();
  virtual void HookUpdateNetworkTime(double network_time);
  virtual void HookSetupAnalyzerTree(Connection *conn);
  virtual void HookBroObjDtor(void* obj);

  virtual void HookLogInit(…); // 2.6
  virtual void HookLogWrite(…); // 2.6
  Virtual void HookLogReporter(); // 2.6
}
```

A hook needs to be activated explicitly:

```
Plugin::EnableHook(HookType hook, int priority = 0)
```

More on Writing Plugins

Much of what I've been talking about is summarized here (except FA):

`https://www.bro.org/sphinx/devel/plugins.html`

Bro package manager documentation:

`https://bro-package-manager.readthedocs.io/en/stable/package.html#binary-bro-plugin-package`

Look at existing plugins:

- Bro packages: **`packages.bro.org/tags`**, filter for **`bro plugin`**
- Bro's source code comes with many built-in plugins

Ask on the development mailing list:

`https://mailman.icsi.berkeley.edu/mailman/listinfo/bro-dev`