Bro vs Suricata

Two Approaches to Network Security Monitoring

Christian Kreibich
christian@corelight.com
@ckreibich
Part 1

Background on Bro
Already covered yesterday
Part 2

Background on Suricata
October 16, 2008 (LAFAYETTE, Ind.) – The Open Information Security Foundation (OISF, www.openinfosecfoundation.org) is proud to announce its formation, made possible by a grant from the U.S. Department of Homeland Security (DHS). The OISF has been chartered and funded by DHS to build a next-generation intrusion detection and prevention engine. This project will consider every new and existing technology, concept and idea to build a completely open source licensed engine. Development will be funded by DHS, and the end product will be made available to any user or organization.
So here's my wish list:

1. Native multithreading.

2. IP Reputation Sharing

3. Native ipv6

4. Native Hardware acceleration support

5. Scoring
A few dozen brainstorm emails later ...
I think everyone is describing Bro

Sorry, I just joined the list so I'm going to be doing some odd quoting from the list archive :) I do want to point out too, that I'm not writing this email to downplay OISFs goals but rather to hopefully guide OISF toward improving an existing opensource project (Bro - http://www.bro-ids.org/ ) that already does much of what is being discussed on this list.
… but 1.5 years later …
DHS, vendors unveil open source intrusion detection engine

Developers look to replace Snort technology, whose backers rebut claims that new Suricata engine is superior

By Jaikumar Vijayan

The Open Information Security Foundation (OISF), a group funded by the U.S Department of Homeland Security (DHS) and several security vendors,
Suricata == Snort++

(2010)
The OISF would “really, really like to displace Snort as the next big IDS engine out there,” Roesch said. “But if you look at [Suricata’s] detection model it is exactly the same as Snort’s,” he added.

Roesch also downplayed the significance of Suricata’s multithreaded architecture and claimed that its implementation would slow the detection engine rather than make it faster. He acknowledged that Suricata’s automated protocol detection capability is useful, but contended that other claimed benefits of the technology, such as IP filtering, are not available on Version 1.0 of the new engine.

Though Suricata has been developed at least partly with government funding, it is unlikely that it can be used in a classified computing environment because it doesn’t allow users the option of hiding the rules they are using for inspecting network traffic, he said. Snort, on the other hand, allows users this option, Roesch noted.

“OISF has wrapped Suricata in some cool computer science concepts,”
OVERVIEW

This version of Snort++ includes new features as well as all Snort 2.X features and bug fixes for the base version of Snort except as indicated below:

- Project = Snort++
- Binary = snort
- Version = 3.0.0-a4 build 235
- Base = 2.9.8 build 383

Here are some key features of Snort++:

- Support multiple packet processing threads
- Use a shared configuration and attribute table
- Use a simple, scriptable configuration
- Make key components pluggable
- Autodetect services for portless configuration
- Support sticky buffers in rules
- Autogenerate reference documentation
- Provide better cross platform support
- Facilitate component testing
Dynamic Application-Layer Protocol Analysis for Network Intrusion Detection

Holger Dreger  
TU München  
dreger@in.tum.de  

Anja Feldmann  
TU München  
feldmann@in.tum.de  

Michael Mai  
TU München  
mmai@in.tum.de  

Vern Paxson  
ICSI/LBNL  
vern@icsi.org  

Robin Sommer  
ICSI  
robin@icsi.org  

Abstract

Many network intrusion detection systems (NIDS) rely on protocol-specific analyzers to extract the higher-level semantic context from a traffic stream. To select the correct kind of analysis, traditional systems exclusively depend on well-known port numbers. However, based on our experience, increasingly significant portions of today’s traffic are not classifiable by such a scheme. Yet for a NIDS, this traffic is very interesting, as a primary reason for not using a standard port is to evade security and policy enforcement monitoring. In this paper, we discuss the design and implementation of a NIDS extension to perform dynamic application-layer protocol analysis. By not assuming the use of a specific protocol analyzer, we can select the correct analyzer for some traffic, which can then analyze the traffic more accurately.

To select the correct analyzer for some traffic, a NIDS faces the challenge of determining which protocol is in use before it even has a chance to inspect the packet stream. To date, NIDSs have resolved this difficulty by assuming use of a set of well-known ports, such as those assigned by IANA [19], or those widely used by convention. If, however, a connection does not use one of these recognized ports—or misappropriates the port designated for a different application—then the NIDS faces a quandary: how does it determine the correct analyzer?

In practice, servers indeed do not always use the port nominally associated with their application, either due to benign or malicious intent. Benign examples include dynamic port allocation in a services discovery protocol like ZeroCONF. Malicious examples include port misappropriation in order to evade stateful firewall inspection. To prevent this evasion, the NIDS needs to be able to determine the protocol of a session that is using a non-classifiable port.
Suricata ~ Snort 3

(2018)
Typical signature

alert tcp $HOME_NET any -> $EXTERNAL_NET any
(msg:"ET TROJAN Likely Bot Nick in IRC";
flow:established,to_server;
flowbits:isset,is_proto_irc; content:"NICK ";
pcre:"/NICK .*USA.*[0-9]{3,}\[/i";
reference:url,doc.emergingthreats.net/2008124;
class:type:trojan-activity; sid:2008124; rev:2;)

Traditional alert log


(Today usually done via Suricata’s “EVE” JSON log)
Part 3

Architecture comparison
Execution flow: **Bro**

- **Process I/O source**
- **Packet source**
  - Run DPI, queue traffic events
- **Broker source**
  - Propagate events
- **Drain event queue**: Execute event handler scripts (alter state, write logs, …)
Execution flow: Suricata

Packet capture → Packet decode → Stream processing → Detections → Output
Execution flow: Suricata runmodes

Packet capture → Packet decode

Packet capture → Packet decode

Packet capture → Packet decode

Packet capture → Packet decode

Stream processing → Detections → Output

Stream processing → Detections → Output

Stream processing → Detections → Output
Execution flow: Suricata runmodes

Packet capture ➔ Packet decode ➔ Stream processing ➔ Detections ➔ Output

Packet capture ➔ Packet decode ➔ Stream processing ➔ Detections ➔ Output

Packet capture ➔ Packet decode ➔ Stream processing ➔ Detections ➔ Output
Keeping state

- In **Bro**: in scripts
  - Tables, sets, ..., with DSL convenience
  - Can be tricky if shared across cluster
- In **Suricata**: in rule language
  - Named bits or counter variables, per flow/host/host-pair

```
alert tcp any any -> any any (msg:"More than Five Usernames!";
  content:"jonkman"; flowint: usernamecount, +, 1;
  flowint:usernamecount, >, 5;)
```
Parsing traffic

- **In Bro**: in C++ or BinPAC
  - Phasing out BinPAC, transitioning to HILTI / Spicy
  - The latter do not depend on Bro, work w/ other tools
  - Goal: safe grammars for syntax and semantics in DSL

- **In Suricata**: in C or Rust
  - Experimental Rust support since 4.0
  - Parsers are Suricata-specific Rust shared libs
  - Provides safety, but separates syntax and semantics
module tar;

export type Archive = unit {
  files: list<File>;
  : uint8(0x0) ; # Null byte
  : bytes &length=511;
};

type File = unit {
  header: Header;
  data : bytes &length=self.header.size;
  : bytes &length=512-(self.header.size mod 512)
};

type Type = enum {
  REG=0, LNK-1, SYM-2, CHR-3, BLK-4, DIR-5, FIFO-6
};

bytes depad(b: bytes) {
  return b.match(/^[^\x00 ]+/); # Strip trailing padding
}

type Header = unit {
  name : bytes &length=100 &convert=depad($$); 
  mode : bytes &length=8 &convert=depad($$);
  uid : bytes &length=8 &convert=depad($$);
  gid : bytes &length=8 &convert=depad($$);
  size : bytes &length=12 &convert=depad($$).to_uint(8);
  mtime : bytes &length=12 &convert=depad($$).to_time(8);
  chksum: bytes &length=8 &convert=depad($$).to_uint(8);
  tflag : bytes &length=1 &convert=Type($$.to_uint());
  iname : bytes &length=100 &convert=depad($$);
  : bytes &length=88; # Skip further fields for brevity.
  prefix: bytes &length=155 &convert=depad($$);
  : bytes &length=12; # Padding.
Parser example: Rust [SPW’17]

```rust
pub struct TlsServerHelloV13Contents<\'a> {
    pub version: u16,
    pub random: &\'a [u8],
    pub cipher: u16,

    pub ext: Option<&\'a [u8]>,
}

pub fn parse_tls_server_hello_tlsv13draft18(i: &[u8]) -> IResult<&[u8], TlsMessageHandshake>
{
    do_parse!(i,
        hv: be_u16 >>
        random: take!(32) >>
        cipher: be_u16 >>
        ext: opt!(length_bytes!(be_u16)) >>
        {
            TlsMessageHandshake::ServerHelloV13( TlsServerHelloV13Contents::new(hv, random, cipher, ext) )
        }
    )
}
```
Extensibility

- In **Bro**: naturally via scripting, also plugins
  - Plugins allow native-code extensions without patching
  - Can add e.g. entire protocol analyzer
- In **Suricata**:
  - Monolithic architecture, usually need to patch ...
  - … unless Lua suffices: can add parsers, detectors, and loggers
Suricata Lua script example: PDF obfuscation [NVISO]

tBlacklisted = {[/JavaScript"] = true}

function PDFCheckName(sInput)
    for sMatchedName in sInput:gmatch"/[a-zA-Z0-9_#]+" do
        if sMatchedName:find("#") then
            local sNormalizedName = sMatchedName:gsub("#[a-fA-F0-9][a-fA-F0-9]", function(hex) return string.char(tonumber(hex:sub(2), 16)) end)
            if tBlacklisted[sNormalizedName] then
                return 1
            end
        end
    end
    return 0
end

function init(args)
    return {[/http.response_body"] = tostring(true)}
end

function match(args)
    return PDFCheckName(tostring(args["http.response_body"]))
end

alert http $EXTERNAL_NET any -> $HOME_NET any (msg:"NVISO PDF file lua";
flow:established,to_client; luajit:pdfcheckname.lua; classtype:policy-violation;
sid:1000000; rev:1;)
Extension example: **JA3 in Bro**

150-line Bro script:
Extension example: JA3 in Suricata
OVERVIEW

This version of Snort++ includes new features as well as all Snort 2.X features and bug fixes for the base version of Snort except as indicated below:

Project = Snort++  
Binary = snort  
Version = 3.0.0-a4 build 235  
Base = 2.9.8 build 383

Here are some key features of Snort++:

- Support multiple packet processing threads
- Use a shared configuration and attribute table
- Use a simple, scriptable configuration
- Make key components pluggable
- Autodetect services for portless configuration
- Support sticky buffers in rules
- Autogenerate reference documentation
- Provide better cross platform support
- Facilitate component testing
Philosophies
Bro is a network monitor

- Nothing intrinsically implies badness
- Perfectly useful *without detecting anything*
- Just happens to be great for deeply behavioral, stateful detections
Suricata is a misuse detector

- Purpose is to detect patterns, usually of badness
- Everything in architecture ties to signature rules
- Has gained Bro-esque features because the community finds these useful, not because they’re natural for the architecture
Culturally different projects

- BSD vs GPL
- Scripting vs “rulethink”
- Research project vs off-the-shelf
- Extensibility vs all-in-one
- Roadmap prioritization
Thanks!