

Playing with snort unified

Index

1. The start
2. The real problem
3. The bug
4. The scope of the bug
5. Proof of concept

The start

After testing OSSIM with snort-2.8* (<2.8.5 and >=2.8) a friend told me that ossim-agent wasn't able to read the logs. It was really strange because after debugging the agent we noticed that the problem was not the ossim-agent. It was snort, because the unified 1 logs were not logging the alerts into the files. So I downloaded the source and my surprise was that if you are using -A fast as an argument in the command line, Snort wont be logging in any other format (using -c /etc/snort/snort.conf). In the beta release (snort-2.8.5) the only mention for "unified" is

```
--
else if (strcasecmp(optarg, ALERT_MODE_OPT__AJK) == 0)
{
    ParseOutput(sc, NULL, "unified2");
}
---
```

and if you are using unified1 it wont be logged.

In the latest release snort 2.8.4.1 its just a switch

```
case ALERT_UNIFIED:
    ActivateOutputPlugin("unified2", NULL);
    break;
```

Ok, so we just commented out -A fast from /etc/default/snort because we really didn't need it since ossim-agent was reading unified 1 format when we were not using the option -A fast.

It has no sense to use -A fast combined with unified when you want fast output.

The real problem

But... We were convinced that the agent was working right, and the events were displayed in the framework but with real traffic we saw that the agent couldn't read the unified logs at all. It was a question of time to read a beautiful Exception of "Bad type" debugging the agent. So I started to look around our parser reinjecting unified logs, analyzing some snort structures like this one (the most important for following a unified log):

```
typedef struct _DataHeader
```

```

{
    u_int32_t type;
    u_int32_t length;
} DataHeader;

#define UNIFIED_TYPE_ALERT      0x1
#define UNIFIED_TYPE_PACKET_ALERT 0x2

```

It goes like this (it's just an overview...):

- 1- Write unified1 file header.
- 2- Write data header(this includes the type)
- 3- if type=UNIFIED_TYPE_ALERT Write UnifiedAlert else if type=UNIFIED_TYPE_PACKET_ALERT Write UnifiedLog and then the packet/pcap information.
- 4- go to 2

So the process is really simple and information is quite easy to parse.

These are the other structures:

```

typedef struct _UnifiedLog
{
    Event event;
    u_int32_t flags;    /* bitmap for interesting flags */
    struct pcap_pkthdr32 pkth;
} UnifiedLog;

typedef struct _UnifiedAlert
{
    Event event;
    struct timeval32 ts;    /* event timestamp */
    u_int32_t sip;    /* src ip */
    u_int32_t dip;    /* dest ip */
    u_int16_t sp;    /* src port */
    u_int16_t dp;    /* dest port */
    u_int32_t protocol;    /* protocol id */
    u_int32_t flags;    /* any other flags (fragmented, etc) */
} UnifiedAlert;

typedef struct _UnifiedLog
{
    Event event;
    u_int32_t flags;    /* bitmap for interesting flags */
    struct pcap_pkthdr32 pkth;
} UnifiedLog;

```

The agent was parsing the file header and some DataHeader, but when injecting a pcap file, in some offset, the type and the size of the next structure was wrong (giving really wrong values for type and size). So I started to read the code for the unified logging process, using the pcap file that was breaking our snort parser and after some work I noticed that there is a bug in the spo_unified.c file, when the Alerts are coming from the Stream5 preprocessor, to be logged at UnifiedLogStreamCallback().

The bug

```
--output-plugins/spo_unified.c 803
SafeMemcpy(write_pkt_buffer + offset, unifiedData->logheader,
           sizeof(UnifiedLog), write_pkt_buffer,
           write_pkt_buffer + sizeof(DataHeader) +
           sizeof(UnifiedLog) + IP_MAXPACKET);

offset += sizeof(UnifiedLog);

unifiedData->data->current += sizeof(UnifiedLog);

if(packet_data)
{
----->SafeMemcpy(write_pkt_buffer, packet_data,
                  offset + unifiedData->logheader->pkth.caplen,
                  write_pkt_buffer, write_pkt_buffer +
                  sizeof(DataHeader) + sizeof(UnifiedLog) + IP_MAXPACKET);

                  if(fwrite(write_pkt_buffer, offset + unifiedData->logheader->pkth.caplen,
                           1, unifiedData->data->stream) != 1)
                      FatalError("SpoUnified: write failed: %s\n", strerror(errno));

                  unifiedData->data->current += unifiedData->logheader->pkth.caplen;
}
else
--+ 825
```

Take a look closely and you'll see that the buffer is overwritten if `packet_data` is not 0, and then the buffer is written to the log file. The fix is really simple. just write in `write_pkt_buffer + offset`, instead of `write_pkt_buffer`.

```
----->SafeMemcpy(write_pkt_buffer, packet_data,...
----->SafeMemcpy(write_pkt_buffer + offset, packet_data,...
```

What values are we reading in our parser? the `packet_data`. Now I know that if I inject packets that snort need to preprocess with `stream5` (to rebuild) and I make sure that they will be generating an alert, the unified log will be corrupted, and the rest of the events in the log will have broken offsets, so the parser that uses this logs to display the alerts will not display the rest of them unless the file gets rotated. Someone could send malformed packets that always break the logs using throwing an exploit to corrupt the logs, or he could even send the attack in the same packet, since it won't have a valid header.

Going back to the raw data... Let's take a look to a backtrace (gdb is my friend).

```
...
#1  0x000000000432ef2 in UnifiedLogStreamCallback (pkth=<value optimized out>,
packet_data=0x92b5050 "", userdata=0x7fffedbd76e0)
    at spo_unified.c:875
#2  0x00000000045a2f3 in GetTcpRebuiltPackets (p=<value optimized out>, ssn=<value
optimized out>,
    callback=0x432c90 <UnifiedLogStreamCallback>, userdata=0x7fffedbd76e0) at
snort_stream5_tcp.c:7956
```

```
#3 0x00000000004319d2 in RealUnifiedLogStreamAlert (p=0x2c58f80, msg=<value
optimized out>, arg=0x1431b60, event=0x1da9944,
  dHdr=0x7fffedbd7780) at spo_unified.c:962
#4 0x00000000004333d6 in UnifiedLogPacketAlert (p=0x2c58f80,
  msg=0x1da95a0 "NETBIOS SMB Session Setup AndX request unicode username
overflow attempt", arg=0x1431b60, event=0x1da9944)
  at spo_unified.c:655
...
```

so let's see a bit more. GetTcpRebuiltPackets calls UnifiedLogStreamCallback with a call back, and here our packet_data is ss->pkt

```
-- preprocessors/Stream5/snort_stream5_tcp.c +7924

for (ss = st->seglist; ss && ss->buffered; ss = ss->next)
{
    callback(&ss->pkth, ss->pkt, userdata);
    packets++;
}
```

and pkt is defined as:

```
const u_int8_t *pkt;          /* base pointer to the raw packet data */
```

uf.. the raw packet data. One of the conditions is that the packet has to be marked as a PKT_REBUILT_STREAM, so we would need to send the data in multiple tcp segments to make sure that snort will be managing it as a rebuilt Stream, in order to execute RealUnifiedLogStreamAlert():

```
-- output-plugins/spo_unified.c +657
if(p->packet_flags & PKT_REBUILT_STREAM)
{
    DEBUG_WRAP(DebugMessage(DEBUG_LOG,
        "[*] Reassembled packet, dumping stream packets\n"));
    RealUnifiedLogStreamAlert(p, msg, arg, event, &dHdr);
}
else
{
    DEBUG_WRAP(DebugMessage(DEBUG_LOG, "[*] Logging unified packets...\n"));
    RealUnifiedLogPacketAlert(p, msg, arg, event, &dHdr);
}
```

More on this later. Let's think a bit more what can be done with this.

The scope of the bug

Let's see if we could make "our own unified packet alert" generating a packet to feed the unified 1 data structures that a parser would read. We need a DataHeader followed by an UnifiedLog, followed by a pcap. And the raw packet that snort would write there would be the ethernet header, followed by the ip header and the tcp header. We need to create a packet with that protocols that would pass as the unified1 structures.

For ethernet packets, it would be (from decode.c)

```
typedef struct _EtherHdr
{
    u_int8_t ether_dst[6];
    u_int8_t ether_src[6];
    u_int16_t ether_type;
```

```
} EtherHdr;
```

The dest address, the source address and the ethernet type. This is 14 bytes, and an unified 1 DataHeader is 8 bytes.

The MAC addresses can be spoofed, so we could generate a packet with the first 4 bytes of the ether_dst with a value of 1 or 2 as if it were an UNIFIED_TYPE_ALERT or UNIFIED_TYPE_PACKET_ALERT

```
#define UNIFIED_TYPE_ALERT      0x1
#define UNIFIED_TYPE_PACKET_ALERT 0x2
```

so the next 2 bytes of the ether_dst and the first 2 bytes of the ether_src would correspond to the "size" of the "UnifiedLog" as in the DataHeader, and the packet would be written in the log as if it were the DataHeader.

```
typedef struct _DataHeader
{
    u_int32_t type;
    u_int32_t length;
} DataHeader;
```

We say type=2, then we have to write a struct UnifiedLog that at the moment is being overwritten by the rest of the ethernet header. The Event structure would be overwritten by the next 4 bytes of the ether_src and 2 bytes of the IP type (0x0800) Let's see what is filling of the event structure.

```
typedef struct _UnifiedLog
{
    Event event;
    u_int32_t flags; /* bitmap for interesting flags */
    struct pcap_pkthdr32 pkth;
} UnifiedLog;
```

```
typedef struct _Event
{
    u_int32_t sig_generator; /* which part of snort generated the alert? */
    u_int32_t sig_id; /* sig id for this generator */
    u_int32_t sig_rev; /* sig revision for this id */
    u_int32_t classification; /* event classification */
    u_int32_t priority; /* event priority */
    u_int32_t event_id; /* event ID */
    u_int32_t event_reference; /* reference to other events that have gone off,
    * such as in the case of tagged packets...
    */
    struct timeval32 ref_time; /* reference time for the event reference */

    /* Don't add to this structure because this is the serialized data
```

```

    * struct for unified logging.
    */
} Event;

```

The 4 last ether_src bytes would fill the sig_generator, and the ether_type (IP 0x0800) would feed half of the sig_id. Now the rest of the Event structure would be filled by the IP Header. The rest of the Event structure is 40 bytes, so we have to see the size of the IPHdr, that is 20 bytes plus the first 6. This is 40 - 26 bytes would be written by the TCP Header in the Event Structure.

```

typedef struct _IPHdr
{
    u_int8_t ip_verhl;    /* version & header length */
    u_int8_t ip_tos;      /* type of service */
    u_int16_t ip_len;      /* datagram length */
    u_int16_t ip_id;       /* identification */
    u_int16_t ip_off;      /* fragment offset */
    u_int8_t ip_ttl;       /* time to live field */
    u_int8_t ip_proto;     /* datagram protocol */
    u_int16_t ip_csum;      /* checksum */
    struct in_addr ip_src; /* source IP */
    struct in_addr ip_dst; /* dest IP */
} IPHdr;

```

Let's see the TCPHeader (32bytes with 12 options). The first 14 bytes of the header would be the last 14 bytes of the Event structure 2+2+4+4+1+1. The th_ack and the th_offx2 and the th_flags would overwrite the ref_time of the Event. Ok, now what?

```

typedef struct _TCPHdr
{
    u_int16_t th_sport;    /* source port */
    u_int16_t th_dport;    /* destination port */
    u_int32_t th_seq;      /* sequence number */
    u_int32_t th_ack;      /* acknowledgement number */
    u_int8_t th_offx2;     /* offset and reserved */
    u_int8_t th_flags;
    u_int16_t th_win;       /* window */
    u_int16_t th_sum;       /* checksum */
    u_int16_t th_urp;       /* urgent pointer */
} TCPHdr;

```

Let's see the UnifiedLog again..

```

typedef struct _UnifiedLog
{
    Event event;
    u_int32_t flags;      /* bitmap for interesting flags */
    struct pcap_pkthdr32 pkth;
} UnifiedLog;

```

The tcp th_win and the th_sum would overwrite the u_int32_t flags of the Unified Log, and the pcap would be overwritten just with the th_urp (the tcp options) and the TCP data!!! We can design a pcap to log the pcap we want to be logged in Unified 1!!! lol what a

coincidence!! The packet timestamp first 2 bytes would be the urgent pointer and the first 2 byte of tcp options would be the rest of the timestamp. The next 4 options would be the caplen size! for example in my case is 08 0a aa 6d that is 0x6daa0a08=1839860232 bytes for our own new pcap. The next options would start filling the pcap. But in fact we don't need them. We could make the TCP window smaller to start the pcap directly with the tcp data. Do your calculations. (I'll make my own calculations with a bit more precision :D).

```
struct pcap_pkthdr32
{
    struct timeval32 ts; /* packet timestamp */
    u_int32_t caplen; /* packet capture length */
    u_int32_t pktlen; /* packet "real" length */
};
```

From this point we could start an attack against unified1 parser/reader applications, and the way they process them.

Proof of Concept

Going back to break the logs. Let's try an example. We can do a proof of concept using the rule "ATTACK-RESPONSES directory listing", that is in the file rules/attack-responses.rules. It just look for the string "Volume Serial Number". With a bit of help from scapy, we will be able to make the job.

- Just recompile snort and add a line like `LogMessage("Overwritting headers\n");` at the line number 809 of `src/output-plugins/spo_unified.c`
- Add in `snort.conf` the line "output unified: finename snort, limit 128" in order to log in unified 1 format.
- Start up snort in from the command line, with something like `./snort -c /etc/snort/snort.conf -l /tmp/ -i eth0`
- And then execute attached file `break_log.py` (You need to install scapy 1 for python). You will see that log message..."Overwritting headers". We have done an "attack" sending the string Volume Serial number and the log was corrupted. If you use for example ossim to get the logs, you wont see that alert in the events, and the agent also wont read the file at all. The size of the malformed alert is filled with the mac, and that would be at least the offset of the next alert inserted. The parser would loop jumping, trying to read a good header type and size until it get a correct one, or the end of file. If you have a parser you can check this.

Now let's go back to design our own alert, using a script like `insert_alert.py` but injecting the content between two packets with more length, to fill the beginning of the first packet data with our own pcap/ethernet/ip/tcp headers :D My calculations were not to bad, I have fixed the size of the pcap with the tcp timestamps but the mac of the "new packet" has 2 bytes less. Anyway it works, but maybe you need a unified 1 parser to check it. The file is `insert_alert.py`. The problem is that one alert is inserted correctly, but the other packet is breaking the log again :D I'm going to fix it so the 2 of them get inserted (we don't want to crash parsers!! heheh ;p). Ok, now the parser read a new record after our first alert, but it have the size bigger than the default config limit for the filesize (128M).. in this case it depends on the parser, if it use the size specified in the header, or if it's hardcoded, because they are usually the same (at the plugin code it's a "sizeof()"). So the log would rotate before the parser can read it. Anyway way know what offset it's going to have the

next record:

if(fwrite(write_pkt_buffer, offset + unifiedData->logheader->pkth.caplen,
So this is the size of the current record: offset + unifiedData->logheader->pkth.caplen. We could fix the scapy packet headers to make the parser jump to the next record continuing the flow!

oh, I forgot.. I have fixed the sig_generator to 1 with the last 4 bytes of the source MAC, but the sig_id is filled with the lp version/tos/length, of the tcp header and it's a bit harder to fix. But this is just a "proof of concept"... with more time we could make tcp packets bigger to insert more than one alert well formed in the tcp data to be parsed correctly, with concrete signature ids and generators, a lot more comfortable to write. Testing it with the ossim-agent it would generate something like this this:

```
snort-event type="detector" date="1535990784" fdate="2018-09-03 18:06:24"  
snort_gid="1" snort_sid="4521992" snort_rev="2388998401"  
snort_classification="104857664" snort_priority="2831188303" packet_type="ip"  
ip_ver="4" ip_hdrlen="5" ip_tos="0" ip_len="60" ip_id="25996" ip_offset="16384"  
ip_ttl="64" ip_proto="6" ip_csum="22063" ip_src="127.0.0.1" ip_dst="255.255.255.255"  
tcp_sport="321" tcp_dport="123" tcp_seq="0" tcp_ack="0" tcp_offset="96" tcp_flags="2"  
tcp_window="8192" tcp_csum="3819" tcp_urgptr="0" tcp_optnum="3" tcp_optcode="1"  
tcp_optlen="0" tcp_optpayload="" tcp_optcode="1" tcp_optlen="0" tcp_optpayload=""  
tcp_optcode="0" tcp_optlen="0" tcp_optpayload=""  
tcp_payload="65686c6f20776f726c6421"
```

The ossim-server doesn't have that snort_sid :) (and I think there is no snort rule with that sig_id).

As you can see the snort_sid, snort_rev, priority and classification are not real, but we could insert more alerts in the same tcp data segment well formed, with the exact values we want for each alert.

The python scripts write a pcap that you can insert directly to snort with -r, but I didn't test it that way. It should work.

Conclusions:

We could:

1. Corrupt the log files.
2. Perform attacks after malformed packets in order prevent that they would get logged/displayed.
3. Make a DOS for the parsers by inserting alerts with header size > than the filesize limit (They would loose a lot of alerts...).
4. Insert a complete falsified attack session by encapsulating many alerts in the malformed tcp packets.
5. We can patch it and be happy with our systems using Stream5 and the rest of the preprocessors :)
6. It's extremely recommended to use unified2 if you're not using it yet.
7. If you still using unified1, use alert_unified or (log_unified), or just unified but patching snort.

This bug wont affect ossim-agent since we were using snort 2.7 and then snort-2.8 patched, but we were really close. Now we are going to start a new parser for unified2 and let's see how great it works :)

Thanks to:

Jaime Blasco and Juan Blanco working on the ossim-agent "arakiri".

Carlos Terrón for his great unified1 parser.

Matt Jonkman and Victor Julien (a pcap generated with the splicer script was the starting point for the scapy scripts).

The OSSIM community and, of course, the AlienVault Team.

Credits:

Pablo Rincón Crespo 07/07/2009

pablo.rincon.crespo@gmail.com