RON DILLEY

making sense of logs

Ron Dilley has spent the majority of the past two decades in the fields of programming, technical support, administration, and security engineering. His involvement in information systems and information security is informed by his vast experience, including several open source projects and work on the 2005 DARPA Grand Challenge. After several years as a UNIX administrator, Dilley moved into information systems security. He currently leads an information security team at a Fortune 500 company.

ron.dilley@uberadmin.com

EFFECTIVE LOG ANALYSIS DOES NOT need to be complex or expensive. Nor do you need to have significant prior knowledge of the data to find anomalies and clear indications of infections. This may sound like the preamble on some glossy product sheet, but don't be fooled. I am not trying to sell you anything. I am trying to encourage you to read on about the log-related tools, techniques, and successes I have had over the last few years.

> To secure ourselves against defeat lies in our own hands, but the opportunity of defeating the enemy is provided by the enemy himself.
>
> —*Sun Tzu, 544–496 BC*

A Short History

I have nurtured a love-hate relationship with logs for almost 20 years. It would be fair to say that my feelings for logs border on obsessive. Instead of spending the following paragraphs on a "logging is good" rant, I present a history of the past few years of my logging obsession and some of the ideas, tools, and techniques that I have worked on.

I was a closet logger for many years. It started back when I was a UNIX administrator running a large call center for a cellular carrier in California. I always subscribed to the idea that it was better to have and not need than to need and not have. I found myself in many situations where solving a problem started and ended with reviewing the copious logs diligently collected from across my environments. I started with awk, moved up to perl, and fiddled around with open source tools like swatch [1] and xlogmaster [2]. As my environments got larger and my logging fervor grew, the volume of log data became unwieldy. It also became harder and harder to make sense of or use all the log data I was storing.

My love of logs did not abate when I switched from system administration to information security. My frustration, on the other hand, grew and grew and grew. I became convinced that there were interesting "things" locked in the massive volume of log data that I was compulsively writing to disk every day. Unfortunately, I could not find any way to set these "things" free. I searched year after year for some way, some tool or script. With every new

product that claimed to solve the logging problem, my heart would leap and subsequently break after grilling the sales people or banging on the product showed that it was not doing anything that my scripts could not already do.

A funny thing happened to me on the way to a security conference that changed everything. It would be more accurate to say that I met another obsessed logger at a conference but for the purpose of this story, it started on the way to the conference. I was thumbing through the schedule for USENIX Security on the plane, looking to see what talks I would attend and noticed that a guy by the name of Marcus Ranum [3] was scheduled to talk about logging. It turned out that he was actually scheduled to rant about logging, but I am getting ahead of myself. It was time well spent. Not because he presented any tools or techniques that blew me away or solved all of my problems, but because he too believed that there were "things" locked away in the logs. I sat in the front row and made a point of introducing myself and plying him with booze and Thai food after class.

We have been collaborating on ways to get at those sneaky little "things" in the logs ever since.

Good Analysis Starts with Reading Your Logs

> Log analysis is a lexical problem.
>
> —*Marcus Ranum*

We focused our work on log datasets stored as ASCII text files. These files were generated using syslog. The best and worst thing about logs produced by syslog [4] is that the format is very open. Other than the PRIORITY and HEADER sections, which have some formatting requirements, the MESSAGE should only contain printable characters, and the whole syslog line should not exceed 1024 bytes. The absence of formatting requirements makes for an interesting parsing problem. For years I used regular expressions to chop up log data. Marcus had a different idea. He built a parser using lex and yacc [5] and spent a few days with Abe Singer at the San Diego Supercomputer Center crunching 10 years' worth of log data. The parser converted raw logs into printf [6] style templates that represented each log line. For example, a simple log line like "Oct 11 22:14:15 mymachine su: 'su root' failed for lonvick on /dev/pts/8" was converted to "%s %d %d:%d:%d %s %s %s %s %s %s %s %s %s". The reason for running this tool against 10 years of data was to get an idea of how much variability there was in the general format of syslog-based log data from a diverse set of UNIX computers. Even with this simple tokenizing strategy, Marcus found about 50,000 different formats across 10 years of data. The graph in Figure 1 (next page) plots the unique templates discovered and clearly shows the increases in new log structures over time. This early version of the parser turned out to be an effective Linux major-release detector, as these spikes aligned well with deployments of new OS versions.

I used the same tokenizing strategy against two years of logs from a Fortune 500 biotechnology company and found less than 8,000 different line formats. This work encouraged me to build a non-regex parser [7] that could extract interesting chunks of data from log data without prior knowledge of the overall log format. In doing so, I came across a couple of interesting and helpful side effects of this parsing methodology. My goal was to make the parser smarter about types of data that I was sure would show up in the logs. To start, this included IP addresses and timestamps.

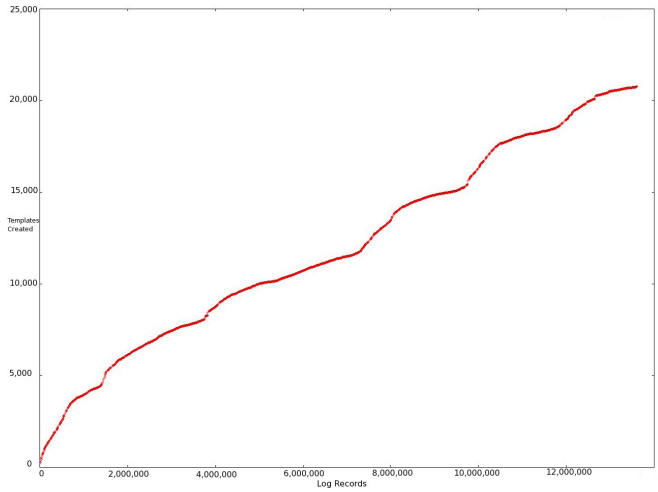

FIGURE 1: PLOT OF UNIQUE TEMPLATES

The utility read logs and converted them into a printf-like template with special tokens for time and date strings as well as IP addresses. White spaces and special printable characters were stored as part of the template. IP addresses and time/date strings were converted to strings. Each non-numeric value associated with a token was stored in a binary tree unique to the template and the token. For example,

```
Feb 5 17:32:18 10.0.0.99 Use the BFG!
```

would be converted into the template "%D %IP %s %s %s!". With the time/date stamp and IP address converted to strings, the five strings would be stored in separate binary trees all associated with the template that the log line matched. At first, I did not realize what I had just built or the interesting and unexpected side effect of organizing the parsed log data in this way. The tool could represent any log line by its template and a unique identifier for each variable. This simple approach was easy to implement, but was inefficient in storing some fields. I experimented with using various compression techniques to save space. A subsequent revision to my code converted recognizable strings such as dates and IP addresses to numbers and encoded them as the numeric value. Treating the fields this way allowed me to see log types I had not seen before, even if some fields had inconsistently variable data, such as the month and day string in a date field. A further optimization was to treat short fields as part of the template so that I did not have to use more space to encode the field than the value consumed in the log. I kept all of the variables in RAM, so I settled for organizing them in binary trees ordered using Huffman encoding [8]. This made it possible to represent variables as a series of bits. Consider this: the example log message above is 39 characters long, and if we assign a two-byte numeric ID to the template (0x01), store the timestamp and IP address as four-byte numbers, and assume that "Use", "the", and "BFG" are the first items in each binary tree, we have stored 39 bytes of data in 10 bytes and the bits required to traverse the Huffman encoded binary tree. My non-regex parser became a tool called logstore [9]. I added some improvements such as assuming that binary trees with only one item could be summarized and moved to the template. Logstore produced 40-to-1 compression of my log data and was much faster than other compression tools that were CPU intensive. The second unexpected benefit of this method came from keeping track of all of the variable strings associated with each log template. By recoding the log line number where each variable occurs, the logstore parsed data format provided a high-speed keyword search capability. I almost regretted encoding some fields as numbers to save space, but good compression was a higher priority than

faster searching at the time. In hindsight, having an option to encode to save space or speed up searches would have been useful.

It was amazing how much fun I could have while taking a stab at regex-less parsing.

Quickly parsing data without significant prior knowledge was nice. We found that non-regex parsing was able to detect new forms of log data that were ignored by regex-based parsers. Unless great care was taken when building the regex parsers, new or edge-case logs were ignored. Using the abstraction of the log templates allowed me to see when a new type of log started showing up by sending the templates into a tool that Marcus wrote called *never before seen* (NBS) [10]. Additionally, I was able to do some simple frequency analysis of the types of logs that were seen without fighting with the complexity of variables.

Alas, it was not enough to satiate my hunger, my need to find those devious "things" that continued to elude detection. A couple of years ago, in response to questions about the efficacy of data-loss prevention systems for my environment, I sat down with Marcus to design a tool that could give me an idea of what we could see on the network beyond simple pattern-matching signatures. Perhaps then I could find those sly little "things" hidden in my log data.

Someone to Watch over Me

> The number of times an uninteresting thing happens—is an interesting thing.
>
> —*Marcus Ranum*

What I needed was a configurable log filter that did not rely on known bad patterns. What came out of our working sessions was a solution with three distinct components. The first components were data collectors that stored network traffic, syslog data, and infrastructure service logs such as DHCP and DNS. The second was a set of parsers that converted the various data sources on the collectors into pseudo-XML format. The last was the tool where all the real magic happened, called overwatch [11]. From a high level, the system works like this:

1. Data collectors generate data which is written to syslog.
2. Syslog data is forwarded to a central repository.
3. Periodically, a batch job kicks off a log parser to convert the syslog data to pXML.
4. The batch job then submits the pXML data to overwatch via a domain socket.
5. Overwatch applies rules and stores scoring information in memory matrixes.
6. Overwatch sends warnings and alerts based on configurable thresholds.

DATA COLLECTORS

Our collectors come in all shapes and sizes. Regardless of how they work, they follow a similar pattern of gathering data and then converting it into text-based logs that are easily parsed. I have built several task-specific programs that extract data directly from the network. I mention several in this article. I have also built scripts, both simple and complex, to convert data from applications such as tcpdump and snort into text-based logs. For consistency and simplicity, I like setting up my collectors to send their data via syslog.

LOG PARSERS

Most of the discussion up to this point has revolved around the evolution of techniques used in our log parsers. Each parser converts the text-based data recorded by the collectors into a pseudo-XML format that is ingestible by overwatch. The name of the game with the parsers is speed: overwatch is a batch-processing environment, so the log parsers determine how fast the data is loaded and processed.

OVERWATCH

Overwatch is a general-purpose filter and pattern detector for arbitrary text-based data. Its simple architecture and configuration belies the power and effectiveness of the tool. The tool maintains n-dimensional matrixes of weighted values. The dimensions, time scales, and weightings are configurable and there is no hard-coded limit to the number of active matrixes. To give you a taste for how overwatch works, I will work through building a configuration to monitor, filter, and alert on DNS logs. Listing 1 shows a record taken from a DNS sniffer [12] that I wrote for use with overwatch:

```
<rec>
<time>Mar 11 18:36:51</time>
<snort dad>123</snort dad>
<srcMac>0:15:c7:c5:22:40</srcMac>
<srcIp>10.131.239.206</srcIp>
<srcPort>32768</srcPort>
<dstMac>0:3:47:de:34:f3</dstMac>
<dstIp>192.41.162.30</dstIp>
<dstPort>53</dstPort>
<dnsId>63706</dnsId>
<qCount>1</qCount>
<qStr0>crl.comodoca.com</qStr0>
<qType>1</qType>
<qClass>1</qClass>
<aCount>0</aCount>
<authCount>0</authCount>
<rCount>1</rCount>
</rec>
```

LISTING 1: A RECORD IN PXML CONVERTED FROM A DNS LOG ENTRY

The pseudo-XML (pXML) begins and ends each log record with <rec> and </rec>. The time attribute is not mandatory; overwatch will use the current time for the record if the field is not present. The <snort dad> field is a unique identifier for the DNS sniffer that sent the record. The other fields are self-explanatory, although we will be using the <aCount> field in another example to discriminate between DNS queries and answers. Next we will build our matrix definition:

```
matrix dnsA
  options {
    alert channel = "/tmp"
    alert recipient =
      "dnsA.alerts"
    warn at 1000
    alert at 5000
    path "/tmp/dnsA"
    rotate hourly
  keyfields {
```

```
        <dnsId>
        <reqIp>
        <aStr0>
    }
}
```

LISTING 2: A MATRIX DEFINITION USED IN OVERWATCH; THE KEYFIELDS DEFINE A THREE-DIMENSIONAL MATRIX.

All matrix definitions begin with the "matrix" keyword. The options section allows you to specify the paths to the data files and high- and low-water marks as well as the time scale for each matrix. The example above puts the data files in /tmp and sets the high-water mark to 5000, the low-water mark to 1000, and the time scale to hourly. This means that data will be placed in matrixes in one-hour increments and that when a value at a given *n*-dimensional position in a given one-hour period exceeds 1000 a warning record will be written, and at 5000 an alert record will be written. The keyfields keyword is where the dimensions are defined. dnsId is the DNS ID associated with the logged query or answer. reqIp is the IP address of the system that sent the query. aStr0 is the first answer string. Listing 2 defines a three-dimensional matrix of dnsId, reqIp, and aStr0.

Now that we have defined our DNS matrix, let's tell overwatch what to do when it reads a DNS log record.

```
records matching {
    <aStr0> startswith "192.168."
} insert into dnsA {
    bump +500
}
```

LISTING 3: AN OVERWATCH INSERTION RULE THAT ADDS 500 TO A MATRIX LOCATION WHEN ASTR0 BEGINS WITH 192.168

Insertion rules begin with the records keyword and have two parts. The first is the match rule and the second is the insert rule. If the match rule is true, the insert rule fires. You can think of the insert rule as a small program that runs against the score values in the matrix. The example above defines a match rule for the first answer string, <aStr0>. If the <aStr0> field starts with the string "192.168.", then the insert rule fires. This is interesting because the DNS logs are being collected from the edge of the network. In general, an external DNS server should not be returning an RFC 1918 address in response to a query for an external hostname. The insert rule adds 500 to the matrix at the position associated with dnsId, reqIp, and aStr0. This rule helped me find some malware that was using DNS answers to issue commands to infected hosts.

The example above is very simple, but don't be fooled. The power of overwatch is in its configuration language. It is limited only by the imagination of the user. Consider the following matrix definition and insertion rule for DNS:

```
matrix dnsConficker
    options {
        alert channel = "/tmp"
        alert recipient =
            "dnsConficker.alerts"
        warn at 1000
        alert at 5000
        path "/tmp/dnsConficker"
        rotate hourly
    keyfields {
```

```
        <reqIp>
    }
}

records matching {
    <aCount> = 0
} insert into dnsConficker {
    bump +1
}

records matching {
    <aCount> greaterthan "0"
} insert into dnsConficker {
    bump -1
}
```

LISTING 4: THIS INSERTION RULE DETECTS CONFICKER DNS FLOODING BY WATCHING FOR MISMATCHES BETWEEN THE NUMBER OF DNS REQUESTS AND RESPONSES.

The matrix rule in Listing 3 is very similar to the example in Listing 2, with the exception that keyfields only has one dimension, <reqIp>. The insertion rules are a bit different. There is no hard-coded limit to the number of insertion rules you can have for a given matrix. For this example I have created two rules. The first adds 1 to the score for a given requesting IP address each time the IP address sends a DNS query. The second subtracts 1 from the score each time an answer to a query is received. This turned out to be a simple way to detect Conficker [13] DNS flooding, as the majority of normal DNS traffic gets at least as many answers as requests. Malware that sends large volumes of bogus DNS traffic looks very different.

The match rules support the usual set of Boolean operators along with string matching and regular expressions. It is also possible to define external files with lists of values to test against. The lists turned out to be helpful in applying adjustments to the scores based on known good or bad criteria such as blacklisted (bogon) networks and domains. A special-match rule calling NBS (never before seen) allows for special insert rules when some value is seen for the first time.

The insert rules support addition, subtraction, and multiplication of fixed values or a value from the log records. I have used this several times. One rule that returned unexpectedly useful information loaded firewall log data into overwatch. It added the bytes out to the score for each destination IP address and port and subtracted the bytes in. This provided a list of the destination/port pairs where internal systems were sending more data outside the network than they were receiving back in. Other than a short list of services like email and VPN connections, most Internet services send more into your network than out of it. Here is output from the rule using the dumpdb overwatch command.

```
% dumpdb –d /tmp/fwByteCounts.2010.03.25.07 dump-all | sort –n –r
13080772    post.craigslist.org|443
619852      www.plusone.com|443
574766      63.240.253.71|443
86940       64.23.32.13|443
85037       www.invitrogen.com|443
79966       miggins.aqhostdns.com|2082
73957       198.140.180.213|443
62292       147.21.176.18|443
```

```
61512       159.53.64.173|443
57494       63.240.110.201|443
```

The dumpdb command was built to aid in tuning the high- and low-water marks for sending out warnings and alerts. As I tuned rules, I found it useful as a stand-alone tool for generating actionable reports about data collected in the overwatch matrixes. The score is the number of bytes out minus the number of bytes in over a one-hour period. This simple rule can detect command and control traffic, unauthorized VPN solutions, and other IP leakage points. Most systems that show up on this list warrant special attention from a security analyst.

A Huge Leap Forward in Log Analysis

It is not my intention to give you a complete dissertation on overwatch and all of its capabilities much though I would love to do so. The point of this high-level teaser is to show why I am no longer frustrated about my logs. I now have a tool that is more than capable of finding those shifty "things," or, as Marcus calls them, "needles in the haystack." In closing, I would like to offer up what I see as the next logical step in the use of overwatch. We call it distrust engineering and it goes something like this.

Systems that interact with "bad" systems are less trustworthy than systems that don't. "Bad" systems exhibit repeating and detectable properties. A system that exhibits one of these properties is less trustworthy than a system that does not exhibit any. A system that interacts with an untrustworthy system is itself less trustworthy. Systems become trustworthy over time.

A negative score shows that a system is untrustworthy and a positive score that it is trustworthy. If we use overwatch to keep track of our trust scores then all we have to do is define what log records represent untrustworthy properties. Once defined, we just need to build parsers that will send the log records into overwatch and we will be able to maintain a near-real-time trust map of all systems in our environment. This ever-changing map will show malware incursions and blooms as well as their retreat as we respond to the incursions. These rules don't need to be complex, and the score adjustments can vary depending on the weight of the event or property. Here are a few attributes that would decrease a system's trust score. I leave the "how-to" for gathering data about each of these attributes as an exercise for the reader.

- Systems in un-trusted countries
- Systems with broken or missing DNS records
- Systems listed in DDNS services
- Systems on malware/spyware black lists
- Systems sending spam or email
- Systems sending packets that are blocked by your firewall
- Systems with new DNS records
- Systems running insecure operating systems
- Systems running unsafe browsers
- Systems with vulnerabilities
- Systems you have never seen before
- Systems flagged by your IDS systems
- Systems that have previously been infected with malware

The above is not meant to be an exhaustive list but merely a primer to get you thinking in the hopes that our email will be flooded with suggestions, rules, and parsers for overwatch.

Conclusion

I have been using overwatch for several years now and my only frustration is that I can't spend more time building and testing new rules and adding additional log sources to the system. My goal was to find a reasonable way to find interesting anomalies in my log data that would help me reduce or remove threats in my environment. I tried many approaches over the years, starting with scripts and moving to log analysis tools and suites both open source and commercial. None of them provided the filtering and detection capabilities that I needed without having significant foreknowledge of the threats. Marcus and I were able to design, build, and implement a generalized detector and filter for arbitrary text-based data. It is useful and effective at detecting anomalous events with minimal prior knowledge or understanding of the event. This required some initial planning and discussion about what log data feeds might be interesting and ways to score the scenarios. As with general log analysis, the best way to do it is to look at the logs, build the rules, and prototype and test them.

REFERENCES

[1] http://sourceforge.net/projects/swatch/.

[2] http://www.gnu.org/software/xlogmaster/.

[3] http://www.ranum.com/.

[4] http://www.faqs.org/rfcs/rfc3164.html.

[5] http://dinosaur.compilertools.net/.

[6] http://en.wikipedia.org/wiki/Printf.

[7] http://www.uberadmin.com/Projects/quickparser/index.html.

[8] http://en.wikipedia.org/wiki/Huffman_coding.

[9] http://www.uberadmin.com/Projects/logstore/index.html.

[10] http://www.ranum.com/security/computer_security/code/nbs.tar.

[11] http://www.ranum.com/security/computer_security/code/overwatch.tar.

[12] http://www.uberadmin.com/Projects/pdnsd/index.html.

[13] http://en.wikipedia.org/wiki/Conficker.