Someone To Watch Over Me

Ronald Dilley, Executive Director, Warner Bros. Entertainment Inc.
Marcus Ranum, Chief of Security, Tenable Network Security, Inc.
(ron.dilley@gmail.com, mjr@ranum.com)

Tags: security, research, data leakage

Abstract

Determining where and how data may be leaking from a network is a difficult problem. In many ways, it is the same problem as intrusion detection: how do you sort a very small event from a vast mass of larger ones? Our answer was to look for less frequent side effects that might indicate a deeper problem, using a tool we developed called “Overwatch.”

This paper describes Overwatch and its configuration, as well as some of the results we attained by running it on two large enterprise networks. We were able to detect some interesting compromised systems as well as the activities of day-zero exploits and the command/control communications of malware like Conficker[1]. In practice, the tool has proven to be useful in a number of ways that we never expected it to be.

Background and Motivation

In 2007, one of the authors was charged with researching “data leakage protection” products for a large biotech firm. Investigating the then-current crop of commercial offerings was not productive: products offered little in the way of predictable reliability and their coverage of avenues for leakage was spotty, at best. Re-purposing of existing intrusion detection system (IDS) architectures was examined and also discarded, because the IDS tend to be oriented toward reporting attack activity rather than auditing data transfer and document movement. Signature-based detection algorithms have one weakness: you have to already know what you’re looking for; in this case we felt we could write matching rules for a number of obvious things like strings in attachments but we’d mostly be detecting “stupid” instead of “naughty.” We were initially attracted to the network-based anomaly detection approach championed by commercial IDS products such as Lancope[2] and Arbor Networks’[3]; these products use a scoring system atop network traffic events, and classify activity as potentially suspicious when it goes above preset point-values. In 1994 one of the authors used a similar approach consisting of shell scripts and diff(1) atop statistics collected by NNStat [4] – work that led to components of the Network Flight Recorder [5]. It was decided to recommend an in-house prototype effort, to get a feeling for the problems inherent in data leakage detection and to gain some experience in advance of the next generation of commercial offerings. We felt it would be useful to have the "speeds and feeds" already figured out, the data already centralized, and a baseline notion of the kind of false positive rates we could expect.

The system design consisted of three components: a syslog(8) collector, a packet vacuum, and a scoring system with the uninspired name, "Overwatch." The packet vacuum was responsible for archival traffic collection of data headed toward network egress points in addition to serving as a DHCP traffic recorder, URL sniffer, and general platform for collecting whatever we thought was worth collecting. TCP traces were collected and stored on a large drive array in a round-robin, indexed by session-Ids and IP source/port/destination. The eventual objective was to have Overwatch examine trace information about the various data collected from the packet and log vacuums as well as other sources, then flag IP source/port/destination combinations as "suspicious." Suspicious streams would be queried out of the packet vacuum and set aside for further analysis that would possibly include passing them to a conventional IDS or a commercial "data leakage prevention" product. In a sense, Overwatch would act as both an anomaly detector in its own right as well as a pre-filter for other detection systems. From this point forward, our paper concerns itself only with Overwatch.

Overwatch

Overwatch is a scoring system that manages databases of weighted event counts. Overwatch maintains multiple detection score-sheets called "matrixes" each of which is updated and monitored on a separate configurable time-scale. There is no hard-coded limit to how many can be supported in a single instance of Overwatch, though there are doubtless practical limits. We never explored the limits of the system, since the design of Overwatch allows an arbitrary number of instances to be running for purposes of parallelism, if needed. In its normal run-state, Overwatch operates as a daemon that is controlled and fed through a UNIX-domain[1] socket, leaving its outputs in structured directories according to its configuration. It is fed via a standalone utility that is both a command shell interface and a data uploader (command: "input") that connects and disconnects for each transaction. The uploaded data is structured in a pseudo-XML[2] (pXML) format by external data collectors – for example, a perl script converting tcpdump output of bootps transactions into pXML records. The example below is from Ron Dilley's DNS sniffer[3], converted to pXML:

[1] It would have been quite easy to make it network-reachable, but we thought that having to engineer security into our security tool would be inviting trouble.

[2] pXML was designed "to avoid having to answer questions like 'why didn't you use XML?' without having to actually use XML."

[3] Passive DNS sniffer (http://www.uberadmin.com/Projects/pdnsd/index.html)

```
<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>
```

As far as Overwatch is concerned, data is nothing more than tagged strings and numbers. The tags are arbitrary identifiers assigned by whatever process creates the input records. In the example above, all the fields that were available in the DNS transaction are marked-up and sent in to Overwatch; everything between <rec> and </rec> is treated as a connected blob of input that otherwise has no structure. We found it was valuable to make our own dictionary of field tags to use for the same values in all inputs, i.e.: <srcIp> , <dstPort>, etc. We will discuss the value of normalizing the field tags later.

In most logging applications, dealing with time is fraught with fiddly details; Overwatch was, unfortunately, not exempted. The tagged <time> field is treated as the absolute time at which the event occurred and is parsed into a clock value. Overwatch cannot be sure that inputs are from the same time window; they might be shuffled, interleaved, or from some time in the past – so it's necessary to derive the event time from the records themselves. In its first version, ninety percent of Overwatch's CPU consumption was spent in time-conversion. Gprof results measured against real data sets pinpointed the problem, and a mechanism for caching date computations was added, which resulted in a forty-fold performance improvement. Dealing with time forced us to make some hard design decisions surrounding handling of back-dated records. Anticipating questions like, "Did Bob's emails transmitted jump dramatically over the last year?" we wanted to be able to seamlessly handle data that was years old being mixed with current data, using the current rules and weightings. This gave the added benefit of allowing tuning against historical data to establish baselines.

Once a data record has been unparsed into memory, Overwatch applies a series of matching rules that add matching records to score-sheet matrixes, assigning them point values based on the record contents. Each matrix is indexed by a set of record attributes, which are used to construct a unique "key" value that holds the cumulative score. Since everything is treated as a string, internally, the keys are simply concatenated with nulls, then stored in dbm(1)-indexed databases that are fronted by read/write caches. The matching rules support a fairly robust set of boolean primitives, based on the contents of the pXML records:

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

A match rule consists of a matching portion which, if true, causes the weighting modifiers to be applied against the matching score in matrix "dnsA." The weighting modifiers represent a miniature program that is run against the score-sheet matrix' values to update them. In this manner, Overwatch can be configured to differentially weight categories or specific types of events. For example, failed login attempts attached to administrative accounts might be treated as +20 point events, whereas failed login attempts in general might score +5 and successful logins +1. The score-sheet matrix for login tracking might consist of pairs of user/host combinations or simply usernames. In the former case, we would be searching for unusual user activity on a fairly detailed level, whereas in the other we'd be looking for unusual activity across all our systems. In practice, we would set up Overwatch to maintain *both* of those matrixes, and separately tune the alert level high water mark to increase or decrease their sensitivity.

It's possible to define multiple rules that manipulate a single score-sheet matrix based on different data inputs or matching criteria:

```
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>
           }
}
```

The sample matrix configuration above shows how the score-sheet index keys are constructed: in this example, they are a concatenation of three fields, which are extracted (if available) from any records that are matched to apply against the matrix. The alert channel/recipient fields construct the output pathnames for warnings that are generated when any data value in the matrix hits a high water mark. The "rotation" value specifies the matrix' "window size" - how often the matrix' score values should be reset. When an input record results in a particular key/value pair in a score-sheet exceeding its high water mark it is written to a matrix-specific alert channel. If it does not result in an alert, the record is stored in a flat file and its offset is recorded in the matrix' database. When an alert is generated, all the records that accumulated the point-score to send a given key/value pair "over the top" are output along with the alert. It is possible, for long-running or large data sets like URL-access or DNS traffic, to turn off the record storing behavior and rely just on summaries. For the data leakage application for which Overwatch was initially built, the alert data were converted into queries to issue against the packet vacuum; a secondary process spun on a directory looking for the appearance of a file of alert data and rolled the file before triggering the query. The administrator received a summary of the alert and its score, the events that led up to the alert (and how they were scored) as well as a link to a pcap(3) file of all the packets still in the packet vacuum that related to the events.

The scoring mechanism in the match rules allows addition, subtraction, multiplication, and adding a fixed per-event "bump" value. Bump values simply add to the score, which is very useful if you want to generate a weighted measure of event frequency. Addition operations are useful if you want to summarize an event; for example, if a log record included the number of kilobytes worth of file attachments, a score might be maintained by adding the attachment sizes. Since all data is stored persistently, there is a lot of potential for generating interesting summary charts from the databases kept by Overwatch.

To simplify the process of maintaining lists for scoring, Overwatch’s matching engine supports a “match list” feature. Lists of strings can be created and referred to as a single set, so that groups of “known ports” or “known strings” can be matched and weighted differently. The same can be accomplished with complicated matching rules, but we felt it was simpler to maintain lists for data such as port numbers, process names, email addresses, or web site names. The matching rule language supports a negation operator, so it is fairly easy to apply different weightings for “member of list” versus “not member of list.” In retrospect, it would have been good to add an if-else syntax to the matching language, to eliminate the need to check list membership twice in the member/not member scenario.

An additional form of event frequency tracking uses what we call “Never Before Seen Anomaly Detection” (NBS)[6]. The idea of NBS is based on the simple observation that the *first* time something happens, it’s always an anomaly. Overwatch maintains a separate NBS database for event keys, and allows a scoring rule to apply a special bump value for the first time an event occurs. We have found NBS to be very useful for things like differentially scoring new MAC addresses, new Email senders, new DNS servers, new IP destination/service port combinations, and so forth. Since Overwatch is intended as a security tool, we’re particularly concerned with early detection of problems, so NBS helpfully acts as a significance amplifier early in a chain of events. In practice, NBS turned out to be one of the more effective parts of the scoring system. After loading a months worth of fully qualified domain names (fqdn) from proxy logs into the system, changing the NBS bump value in our matching rules to a value larger than the high water mark presented us with a short list of new domain names per day. A quick glance over the list identified high risk destinations visited for the first time by internal hosts.

```
mjr@lyra-> ./dumpdb -d \
   /tmp/dnsQ.2009.03.11.18.index \
   summarize +1
      697 fe80::211:25ff:fe3e:9773
      657 fe80::211:25ff:fe41:b09
      624 177.185.151.198.in-addr.arpa
      463 fe80::211:25ff:fe41:a5e1
      417 wbamanp001
      341 ldap._tcp.WBCCTV.COM
      314 sb.r1.rdc.itiva.net
      188 172.222.234.207.in-addr.arpa
      158 ejlaptop.warnerbros.com
       80  sb.r3.rdc.itiva.net
```

The score-sheet matrix can also be used as a summarizer for log data, and can be quickly dumped in a variety of ways. Since the Overwatch engine may keep dirty data in its memory, there is a command line interface (cli) command that can be used to pause and flush the engine, preventing inconsistencies while queries are performed. A database dumping utility with the prosaic name of “dumpdb” converts databases into plain text or pXML, with a variety of sorting and field-merging options. The resulting information can be used to plot trends or to quickly retrieve “top N” events, i.e.: “top 20 web sites visited through the proxy server” and “top IP addresses accessing ports that we have not defined as typical Internet services.” This tool turned out to be very helpful in tuning both the match-modify rules as well as the high-water marks to optimize the filtering.

Alerts and warnings include summaries of the information that produced a score:

```
WARNING: Sat Apr 11 18:40:41 2009
reference #4088 first seen in db: Sat Apr
11 18:40:41 2009
cumulative score 10505 with 3 events
Key Attributes used in computing this
event {
    DNSID=48769
    REQIP=10.131.239.206
    ASTR0=10.0.132.254.210.in-addr.arpa
}
```

Based on the reference number included with the alert, the database dumper can extract all the events related to the alert, along with their score values. Sometimes this is a great deal of information, which is why it proved practical to add the summary and data-coalescing options to the database dumper. For example, based on the warning above, we might query for a summary breakdown of connections between the requesting IP address and the destinations of its requests:

```
16576004       10.10.10.10
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
```

We are not under the illusion that the system is completely automated; unlike most signature-based intrusion detection or data leakage systems, we assumed from the outset that experienced analysts were going to be looking at the data produced on the backend, and favored simple text tables wherever possible. A separate research project was launched to use large-scale scientific visualization software to examine trends in the data produced by Overwatch.

Practical experience

Overwatch was originally purposed to detect data leakage, but we managed to also detect malware. Conficker[4], for example, left a rather large footprint in the DNS usage matrix during its regularly scheduled updates. An infected host generated 50,000 predicable domain names and executed DNS queries against random chunks of them, each day. This pattern showed up on our NBS rules as singular hosts querying for large numbers of never before seen domain names. Though this was helpful in identifying possible Conficker infected hosts, it still required some time each day to filter through valid NBS domains that matched the 4 to 9 character length and any one of the 110 TLDs the 'C' variant generated. The domain names are pretty easy to spot in a list as shown:

```
pojusjpvg.pl
yhgxgh.com.ar
phexuk.com.tr
xenryfcy.com.jm
xqnes.com.bo
akfzy.me
pfbtagrli.cx
ipodfx.com.ai
btwhzea.bo
cfatn.nl
wzbx.tw
urwgnc.com.fj
dbhmuven.tn
```

One of the authors added an additional rule to reduce the matrix score if there was a successful DNS answer. This was done using a positive bump rule that matched an outgoing DNS request, and a negative bump rule that subtracted the points again when it saw a matching response. Negative bump rules have proved very helpful in refining some filters to de-emphasize "known good" patterns such as query/response and transmitted/received

[4] Also known as Downadup, Downup, and Kido

acknowledgements. As the vast majority of DNS requests for Conficker domains fail, this immediately improved the signal to noise ratio without the need for manual review. A further improvement which we considered (but did not implement) was to add another bump rule for any domain that matched one of the Conficker TLDs.

Another simple Overwatch rule that has proved very effective at detecting malware command and control traffic applies “number of bytes out” minus “number of bytes in” on all network communications. The matrix definition includes the source IP, destination IP and destination port. In corporate environments, typically there are very few situations where the amount of data leaving the organization is greater than the amount entering. Most exceptions fall into the category of business to business data transfers, encrypted tunnels and e-mail. Most of these transactions happen on well-known sets of source or destination IP addresses. When we first deployed this rule we were surprised to discover that three out of four destination addresses had no reverse DNS records and of those, the majority easily mapped to known or suspected malware-friendly internet service providers. This simple rule continues to be effective at identifying command and control traffic for zero-day malware.

Overwatch has shown its usefulness not only in identifying malware, it is also effective at fulfilling Sarbanes Oxley (SOX) related reporting requirements. As all logs in our environment are forwarded to a central logging repository we built a set of rules to process authentication success/failure logs for UNIX and Windows systems. The first version bumped the score on login failures for a matrix of hostname and login name with an NBS rule to amplify the significance of new host-login combinations. The output was a daily report of systems where there were a large number of login failures.

The following examples show the simple login failure matching rules for Windows logs:

```
records matching {
       <source> exactly "Authentication"
} insert into logins {
       nbs bump +1000
       bump +1
}
records matching {
       <eventid> not "540"
          and
       <eventid> not "528"
} insert into logins {
        bump +1000
}
```

The following is a report generated using dumpdb:

```
208208  AGHSYILS3SD\phil.g.xxxxxx
104104  AGHSYILS3SD
13013   GREST01
23023   DBF345\SVC
23023   DBF345
20020   CFWPURP-TSPWV09\Administrator
23023   DBFCF01
10010   CFWPURP-TSPWV09
```

We implemented “too many failure” detection by pairing a login failure rule with a positive bump value against a login success rule with a matching negative bump value. A failure matched with a success cancels itself out, but unbalanced failures trip an alert. Overwatch supports both high and low water mark alerting; we were able to use one matrix to track too many successful logins as one problem, and too many failures as another.

In terms of data quantities that Overwatch handled at our installation, we averaged about 50Gb of data daily consisting of approximately 10Gb of firewall logs, 2Gb of SMTP logs, and 40Gb of web proxy logs. This load was handled with a single instance of Overwatch with all the data being fed into it sequentially. We didn’t put any effort into worrying about performance other than during the design phase of the system, since the data handling was parallelizable. Because the majority of Overwatch’s CPU time is presently spent in unparsing dates, we expect that a faster/simpler date-unparsing routine would be worth developing; most log records come in time-clusters and caching the date/month/year/GMT offset computation could speed things up. The input parsers used to select fields from the static logs handle on the order of 100,000 log-lines per second. In general, we were pleased with the performance of the system; it was intended for batch mode operations but offered near “real time” results.

The biggest flaw in the implementation of the system was a result of Overwatch’s design as a batch mode process. The daemon sits in a run state and periodically expects connections to feed it new data. While it’s handling its input, the databases are cached in memory and the server is “busy” and will block the submission process if something attempts to feed it another data set. In a batch mode model, this works acceptably, because the inputs automatically serialize, but it also has the effect of blocking queries or server commands until the server is free to handle them. Having to wait several minutes for the server to chew through a dataset can be frustrating while you’re in the process of tuning the event weights. If there were ever to be a Version 2.0 of Overwatch, it would be nice to rethink the mechanisms whereby data flows through the system, though we think the overall feature-set is pretty complete.

Lessons learned

The effort provided tangible data relating to volume and types of data we could extract from our network, logs, and analysis. We also got a useful baseline for the signal to noise ratio of our security event logs, and the effectiveness of our filtering, and found Overwatch to be a flexible and powerful tool for quickly exploring “what if?” scenarios. Having the infrastructure in place where we could quickly set up rules to examine relationships between event frequency and security/data leakage events turned out to be particularly valuable. Ron’s total time between asking himself “could we detect Conficker worm by looking at DNS rates?” and having a functioning ‘Conficker detector’ was about 2 hours with the majority of the time spent loading old DNS logs into Overwatch to seed the NBS data. More importantly, that detector generalizes and might find other interesting similar but unrelated types of traffic.

If we were re-implementing the system, we would now know enough to invest more time in making aspects of Overwatch and its operations faster. We found that the work-flow had slow spots, specifically around writing parsers to translate and upload data, as well as testing hypotheses for places where we might detect something. It generally took several hours to set up a matrix and debug matching rules, with subsequent testing runs to make sure the data was being correctly parsed and input. Sometimes the work wouldn’t pay off, which was frustrating, and it was hard to determine whether it was a result of configuration or simply that there was nothing to find.

Now that we have a better idea how a system like Overwatch is used, we would prefer to have made the daemon process capable of handling multiple streams of input, and would have embedded the query/dumpdb process into the daemon, itself. That would allow the daemon process to cache index information without having to worry about consistency issues running dumpdb against the files directly through the filesystem. As it stands, debugging an Overwatch configuration requires a great deal of starting and stopping processes, though much of these problems could be overcome by writing scripts that managed the details. Since Overwatch only tries to do one thing at a time, it's impossible to query it while uploading data to it without the query blocking until the upload completes; this exercises the patience of the analyst more than is desirable. If there is a future version of Overwatch, we intend to add some statistical capabilities into the matrix structures. It would be interesting, for example, to compute moving averages, or to be able to keep summaries of historical trends in a matrix (e.g.: what is the average number of emails sent per host every month for the last year?) We also think it might be fruitful to have one matrix be able to insert data into another, producing a sort of massive roll-up report over time. It would also be interesting to incorporate external blacklists such as the Google malware hosting site list or spam sending list; we are investigating ways of incorporating queries through external APIs although it might make more sense to simply do those queries as part of producing the pXML to upload to Overwatch.

In our initial design discussions we realized that the problem we were dealing with was that there was valuable data hidden in our log data and the volume, diversity and lack of foreknowledge of the data prevented us for getting to the tasty 'needle in the haystack.' Overwatch was our attempt at building a filter that would allow us to get at the needle. Most importantly, this experience re-confirmed for us something that we already knew: in log analysis there is no amount of conjecture that can be engaged in that is as valuable as a few hours of actually looking at your log data.

Conclusions

A generalized detector and filter for arbitrary text based data is useful and effective at detecting anomalous events with little or no prior knowledge or understanding of the event. We found that by sitting around hypothesizing a few "what if" scenarios, we were able to relatively quickly prototype working anomaly detectors; sometimes they paid off, sometimes they didn't. As we had hoped, addition and subtraction turned out to be as useful as complicated statistics.

Availability

Overwatch is available in source code form for noncommercial / research use, though the documentation for it is sparse. If there is enough interest to warrant documenting it, we will do so. Contact one of the authors if you're interested.

References

[1] An Analysis of Conficker's Logic and Rendezvous Points, Phillip Porras, Hassen Saidi,

and Vinod Yegneswaran, http://mtc.sri.com/Conficker, February 2009

[2] Lancope Stealthwatch™ http://www.lancope.com

[3] Arbor Networks Peakflow ™ http://www.arbornetworks.com

[4] Robert Braden and Annette DeSchon, NSFNET Statistics Collection System NNSTAT, USC Information Sciences Institute, December 1992

[5] Implementing a Generalized Tool for Network Monitoring, Ranum et. al., Proceedings of the 11th USENIX conference on System administration, 1997

[6] Never Before Seen Anomaly Detection, Marcus Ranum, http://www.ranum.com/security/computer_security/code, August 2004