Global Information Assurance Certification Paper

Matthew Shaul
GSEC version 1.2d

Using Swatch to Utilize Your Logs.

When securing a Unix system, the admin has at his (her) disposal many tools, many of which are third party downloads. However, as pointed out in Practical Unix and Internet Security, one of the most basic and useful tools is a system's own log files, "Log files are an important building block of a secure system: they form a recorded history, or audit trail, of your computer's past, making it easier for you to track down intermittent problems or attacks. Using log files, you may be able to piece together enough information to discover the cause of a bug, the source of a break-in, and the scope of the damaged involved. ...Those logs may be exactly what you need to rebuild your system, conduct an investigation, give testimony, recover insurance money, or get accurate field serviced performed.".[1]

From the excerpt above, you might be persuaded to log everything possible. Coming up short during an investigation or during the process of rebuilding a critical machine because the logs are short on information is the last thing an admin would want to experience. So if logging system and application data is so critical at many levels, why is it that it's often not done correctly? I believe Matt Morton says it most succinctly in his SANS paper, Logging and critical logs files: the Decision to Effectively and Proactively Manage System logging and Log Files ,located at http://www.sans.org/infosecFAQ/securitybasics/logging.htm. He gives two reasons: lack of time and lack of understanding how to do things differently.[2] Let's assume the admin has taken the time to be proactive and has familiarized himself with setting up a centralized logging server so multiple network devices and systems are all sending their logs to the server. But having the systems write to the log server is not enough. To fully avail ourselves of the power of logging requires us to review the logs in a scheduled and timely manner. Not reviewing the logs often, preferably in real-time but at least daily, gives an intruder the time to cover his tracks, alter your system, and take advantage of the machine before you notice. However, consider a possible consequences of the security policy of having systems logging extensively. It will require the admin, or someone, to scan the subsequently huge logs in order for the policy to have any effect. Soon the admin will be faced with looking over massive logs with line after line after line of mostly familiar and uneventful messages. The chance is greatly increased of the admin making careless mistakes by overlooking critical messages.[3]

Consider one of my home machines, a typical RedHat 6.2 box running not much else other than ipchains and portsentry as my internet gateway on a DSL line. A quick look in /var/log/messages shows that a day's worth of logs averages about 3 MB, or over 20,000 lines. And this is just /var/log/messages. Imagine having to also go through an xfer log, mail log, samba log, etc. Imagine the production environment with the central syslog server mentioned above which manages the logs of multiple servers, routers and other network devices. With this amount of data, the lack of time, as well as boredom and carelessness become the admin's worst enemy. An overstretched admin may come to

view as a burden the line after line of system information often separated by a mere 1/100th of a second. Proper log checking will surely become one of the first areas affected by corner-cutting as a method to save time. At worst, the logs are ignored, possibly deleted and never scanned. Or perhaps compressed and stored with the admin's intent to make time to review them later - possible, but not likely. For proper system security, this situation is nothing short of disastrous.

But even in the case where the admin has ample time and ability to parse the logs, the effort may prove to be ineffective because the log review is in reality a reactionary measure considered critical only after a user or admin notices something "odd". Consider the admin who shows up for work at 7 am and diligently begins combing through the logs from the night before. He nervously puts his coffee down as he notices that the logs indicate an unauthorized user su'ing to root, or a telnet session when he thought that the telnet port was not open. Suddenly the ability to scan back through previously written lines, or even old logs becomes crucial. Looking at the timestamp, he notes that the offense occurred at 7 pm the night before. Effectively, the bad guy has a 12-hour head start. Surely not an insurmountable task, but using our figures from above, there are perhaps 10,000 lines of output that the admin must now quickly, but ever-so-carefully go through. Wouldn't it have served the admin well to have been notified immediately upon the occurrence of the offense? Of course it would have. And wouldn't it have been much more efficient if we had a method for our admin to go through the logs after much of the unneeded or mundane lines were parsed out? And what if we had a simple procedure that allowed the admin to quickly and efficiently go through logs multiple times using different search criteria, or an efficient method to parse old logs that had been archived? Well there is such a tool, mentioned in Matt Morton's paper, that allows all of this and more: Swatch- the Simple Watchdog. I would like to take the rest of this paper to point out some of its uses and benefits.

Swatch was developed by Todd Atkins as a tool, written in Perl, that allows admins to cull through enormous amounts of logged data without being overwhelmed. Atkins states four design goals in writing Swatch:

1. Configure the program is such a way that it would only take a few minutes to teach any system administrator how to use it.
2. Have a simple set of actions that could be performed after receiving certain types of information.
3. Allow the user to define their own actions if they like, and allow them to use parts of the input as arguments to the action.
4. Once Swatch is running, it should be reconfigurable on demand or after a specified interval without having to stop and restart the program by hand.[4]

All of the design goals were met, but number two, in particular, allows us to use Swatch in conjunction with our logs and with other tools to enhance our ability to secure our machines. For instance, used on a machine running portsentry, we can configure Swatch to page the admin if the phrase "attack alert" is written to a log that we have Swatch

monitoring. Or perhaps we'd like to be emailed if a user has unsuccessfully tried to log in too many times. Later in the configuration section we'll discuss these options further.

Now to begin using Swatch, we need to download it from any one of a number of sites on the Web. It's official home is ftp.stanford.edu/general/security-tools/Swatch. Additionally, if not already on the system, we need three Perl modules: Date-Calc, Time-HiRes, and File-Tail. These also can be found in numerous places on the Internet, with http://www.perl.com/CPAN-local/modules/01modules.index.html being one of them. After downloading Swatch and the Perl modules, installation is quite painless:

unzip and untar each of the Perl modules, for example the Date-Calc -

```
tar zxvf Date-Calc-4.2.tar.gz
```

cd into the newly created build directory and configure the module for building -

```
cd Date-Calc-4.2
perl Makefile.PL
```

make the module, test it, and install it -

```
make
make test
make install
```

cd back to the directory holding the remaining modules and make them similarly.
Next, we install Swatch by following the same procedures we used for the modules:

```
tar zxvf swatch-3.0b2.tar.gz
cd swatch-3.0b2
perl Makefile.PL
make
make test
make install
```

Assuming no errors, that's it. We are nearly ready to begin configuring Swatch. But before we do, we need to become familiar with its capabilities and how it interacts with our machine.

Now as we'll see, we can use Swatch in a couple of different ways; for example we can watch our logs in real-time or review previously written logs in batch mode. In either case, the common denominator in using Swatch is that it parses written logs, and therefore we need to be at least cursorily familiar with the /etc/syslog.conf file since this

is the main logging configuration file for the machine. (*note: This example is for a RedHat 6.2 machine.)* Generally you can keep the default setup if after looking through your available logs, you feel that the information you're interested in reviewing is presently being logged. However, if the information is not there or if you'd like to move the output to another log file, you'll likely need to modify /etc/syslog.conf. This is when it may be best to implement a plan to determine what it is we want Swatch to help us accomplish. Let's consider a simple three step process used by Lance Spitzner in his paper, Watching Your Logs:

1. First, define what you want to know.
2. Determine which logs contain this information.
3. Lastly decide what will be considered a trigger in the logs.[5]

Let's take my home machine mentioned earlier to go through the three steps. First we define what it is we'd like to know. I want to know when portsentry believes that my machine is possibly under attack and I'd also like to know about any unsuccessful su attempts. Of course, in reality we'd like to know much more, but let's start with this as an example. Second, we determine to which logs this information is written. Checking my portsentry_config.h file tells me that portsentry will by default use syslogd and will use a "notice" level. So checking /etc/syslog.conf informs me that notice entries will be logged to /var/log/messages. Running *info su* informs me that su can use syslogd, so I'll check syslog.conf again and find that /var/log/messages is the log to check. Lastly, for step three we need to define from /var/log/messages what precisely is logged that will trigger our alert when information of interest to us is written to the log. We can accomplish this in a couple of different ways. One, we can search the logs for a portsentry-perceived attack and an unsuccessful su and make note of the logged entries. Or we can open a terminal window and perform an unsuccessful su and then go to another machine and initiate a portscan to trigger portsentry. Returning to our original machine and viewing the logs will show us exactly what was logged.

After these three steps, we are ready to begin creating our Swatch configuration file. You'll see that getting a usable initial configuration file up and running can take only minutes, but fine tuning the file may take significantly longer. Swatch by default will use ./.swatchrc as its configuration file. However we may create multiple configuration files based on what it is we want Swatch to search for in that particular instance. The configuration file consists of regular expression patterns that must be parsable by Perl, and the actions to take upon finding a match to these expressions. The actions taken are determined by the keywords: *watchfor* and *ignore* (the keyword *waitfor* is often listed in literature as an option but is no longer in the man pages and seems to have lost its functionality). The actions to match with these keywords are: *echo [=mode], bell [=N], exec=command, mail [=address:address...], pipe=command, write [=user:user...], throttle=options*. The function of most of these actions is evident, but we'll step through a few to demonstrate how they might be used. For this example I'll assume we are going to use Swatch to monitor /var/log/messages in real-time, similar to opening a terminal and tailing the log, but having Swatch only write to terminal those lines that match our

criteria. From step three above we find in /var/log/messages the lines that trigger our requirements:

```
# this is the log entry for the unsuccessful su attempt
May  3 07:29:56 rhs PAM_pwdb[32680]: authentication failure;
mshaul(uid=500) -> root for su service

# this is the log entry for the successful su - we'll use this in a later config
May  3 07:29:56 rhs PAM_pwdb[32696]: (su) session opened for user root
by mshaul(uid=500)

# this is the log entry for what portsentry believes is an attack
May  1 12:09:12 rhs portsentry[597]: attackalert: SYN/Normal scan from
host: 200.68.61.194/200.68.61.194 to TCP port: 53
```

Use your favorite editor to open .swatchrc. Comments are preceded with a # and are ignored by Swatch. The first thing we need to do is define an expression and action for our requirements by using part of the logged message as our matching expression:

```
#Here we define our unsuccessful su attempts.  The actions we'll have
Swatch associate with
#these attempts are:  echo to the terminal in bold lettering, send an email.
watchfor = /su service/
echo = bold
mail = mshaul@kvw.com

#Here is our portsentry entry.  Again, we'll be emailed, echoed red, and
ring the machine's bell 3 times.
watchfor = /attackalert/
echo = red
bell = 3
mail = mshaul@kvw.com
```

In a most basic form, that's all that's required. It's not much of a fine-toothed security comb, but it is functional. First we'll test it and then begin to tune it.

To test, from a terminal window run:

```
swatch -c ./.swatchrc -t /var/log/messages
```

The *-c* option tells Swatch which configuration file to use, and the *-t* option tells Swatch to run in real-time (tail the file) and of course, /var/log/messages indicates which log we'd like monitored. Now from another terminal window, purposely enter a wrong password while su'ing to root to test the first stanza, and then portscan the machine to test the second. Returning to examine our Swatch terminal we find:

```
May  1 12:09:12 rhs portsentry[597]: attackalert: SYN/Normal scan from
host: 200.68.61.194/200.68.61.194 to TCP port: 53
May  1 12:09:12 rhs portsentry[597]: attackalert: Host 200.68.61.194 has
been blocked via wrappers with string: "ALL: 200.68.61.194"
May  1 12:09:12 rhs portsentry[597]: attackalert: Host 200.68.61.194 has
been blocked via dropped route using command: "/sbin/ipchains -I input -s
200.68.61.194 -j DENY -l"
May  1 12:44:55 rhs PAM_pwdb[515]: authentication failure;
mshaul(uid=500) -> root for su service
```

The text for the su entry would be bolded for the attackalert red, the bell would have rung nine times (3 for each attackalert line), and I would have received 4 emails (one for each entry). A quick way to eliminate some of the duplicate mail would be to use more specific matching criteria. Additionally, for security purposes, we probably shouldn't limit ourselves to only the unsuccessful su attempts, rather we would also like to know of successful attempts in case the user who at first was unsuccessful does finally get root. Our new config file is:

```
#Here we define *all* su attempts.  The actions which we'll have Swatch
associate with
#these attempts are: echo to the terminal in bold lettering
watchfor = /su/
echo = bold

#Here we match authentication failures. We are looking primarily for
failed su's
#but we'll take any failed authentications.
#We will be emailed about these.
watchfor = /authentication failure/
echo = yellow
mail = mshaul@kvw.com

#Here we match the second and third lines of the above attackalert.
#They will be echoed, but we won't be emailed.
watchfor = /attackalert: Host/
echo = red
bell = 3

#Here our attackalert matches "attackalert.*scan" because we never know
#if it will be a SYN/Normal scan or some other type.  The message will be
#emailed, echoed red, and will ring the machine's bell 3 times.
watchfor = /attackalert.*scan/
echo = red
bell = 3
mail = mshaul@kvw.com
```

```
#This next stanza is important.  It acts as a kind of fail-safe.
# It says, if it's not specifically mentioned above, match everything else.
watchfor = /.*/
echo
```

Now when we run through our tests we will cut down on the amount of email per portsentry incident that we receive, we will catch all incidents of su'ing, and our catchall stanza at the end will put forth an immense amount of data. This last stanza is important; it will catch everything not specifically mentioned above it in the config file. But doesn't this just add to the problem we're trying to solve of too much information for us to scan? Initially yes, but this is when the other key word, *ignore*, becomes invaluable. As we go through Swatch's output, we find entries we feel are safe to ignore and enter them into the config file with the *ignore* keyword. For example, perhaps our default gateway is running RIP and is broadcasting updates. We can eliminate these from Swatch's scan. We may also want to ignore the netbios packets that we're denying from the Windows machines on the networks. We could decide to ignore DNS responses from our DNS server. Additionally, we may notice items from the catchall to which we need to pay closer attention. For example, assume that we have a print server that is logging information about a printer that intermittently goes bad. When it does, it logs messages every two minutes. We want to know when the printer has a fault, but we don't need to have Swatch put a message out every two minutes, so we'll use the keyword *throttle*. This tells Swatch to react to the first matched entry but to ignore similar messages for a length of time that we specify. The entries below would be added to our existing configuration file.

```
#Let us know about the faulty printer once
#and then not again for an hour.  Hopefully we'll have it fixed by then.
watchfor = /printer/
echo = blink
throttle = 1:0:0

#Ignore RIP broadcasts
ignore = /PROTO=17 192.168.1.1:520/

#Ignore denied netbios packets
ignore = /DENY eth0 PROTO=17.*:(137|138|139)/

#Ignore DNS traffic if it comes from our DNS server
ignore = /PROTO=17 192.168.1.3:53/
```

We continue to review Swatch's output and make changes to our config file in this way. Our catchall ending stanza, /.*/ , ensures that Swatch will continue to report logged events that we've not yet encountered. This will allow us to decide at that time whether we need to associate a Swatch action with the event or to ignore the event altogether. It

may be tempting to try to write a config file to only to watch for events to which we already know we want Swatch to react. But it would be more prudent to always use the catchall stanza because there is simply no way for us to know beforehand all of the possible messages that we might find interesting, or worrisome. So how do we decide whether to watch or ignore entries? The recommended practice from the excellent book, Building Internet Firewalls, instructs us to view log entries as being in one of these three categories:

- *known to be OK* - These can be safely ignored. Just be sure to know exactly why and when these are logged.
- *known to be dangerous* - Definitely logged and likely associate an action such as an email with them.
- *unknown* - These need to be examined and investigated. For example, search security websites for new vulnerabilities, etc.[6]

This will be an ongoing process as new unknown messages are viewed. Each new unknown message, after review can be classified as a watch or an ignore.

Another keyword, *exec*, allows us to execute a command when a pattern is matched. The earlier version of Swatch, 2.2, had a handy utility named call_pager.pl that could be used with the *exec* keyword. So if a critical pattern was matched, Swatch would use the machine's modem to page the admin. We could also pass a number code (its relevance only important to us) indicating the severity of the situation if we had more than one paging event in the configuration file. This allows for more immediate response than perhaps email. However this utility doesn't appear to be included in the latest versions of Swatch, but it can still be brought over from the older versions, which can be downloaded from ftp://ftp.stanford.edu/general/security-tools/swatch/OLD/swatch-2.2.tar.gz . Untar the package and look for it in the utils directory. The usage would be:

```
#Alert us about kernel problems. Page the admin with situation code 111.
watchfor = /(panic|halt)/
echo = blue
bell = 6
mail = mshaul@kvw.com
exec= "/usr/local/sbin/call_pager.pl 5554321 111"
```

Let's consider another useful application of Swatch that can help us in our security procedures. Even with a finely tuned configuration file, we never know what kind of information we may eventually want to extract from our logs that we haven't already pulled with our existing configuration file. For example, perhaps we've decided to ignore successful user login messages from the file server rather than have Swatch report each one. But now management suspects nefarious doings by one of the users. We have all of our archived logs and we have Swatch to go through these logs; we just need to put together a quick temporary configuration file tailored for our specific needs.

```
#Find anything in the logs having to do with user jsmith
watchfor = /jsmith/
echo
```

Perhaps we've trusted another subnet to use services on our machine with our /etc/hosts.allow file and now we find that some of their machines have been compromised. We'd like to go back through our logs and examine activity from this subnet, so we write a configuration file.

```
#Find activity from this network
watchfor = /192.168.1.*/
echo
```

In both cases, we write the config file, say "swatchtemp", and apply it to any of our logs, even the archived ones. The -c switch points to the config file we want to use and the -f indicates the log file. We can of course redirect output as we have here:

```
swatch -c ./swatchtemp -f messages.1 > Mgr_Report
```

After Swatch returned information from our search, we could further refine our config file to look deeper into the user's movements on the machine or about any activity from the trusted network.

Let's add one last bit of functionality. So far we have Swatch parsing our old logs on an as-needed basis and monitoring our current logs in real-time, emailing and paging us if needed. But we can not be at the monitor 24/7 to see unknown items being caught by our catchall stanza, or to see items that are only echoed to the screen and not emailed to us. Surely we'd need to implement a method that allows us to view all items that Swatch outputs, especially after we believe we have fine tuned the configuration file to the point where we feel only extraordinary items will be caught by our catchall stanza. A solution could be to use a script that has Swatch parse a log and email the output to us. The script below will do this and will have Swatch not rescan lines that it scanned in the previous time period. For example, if we run the script nightly, we are not notified in today's email about the same events we were notified about yesterday. This email would be in addition to any email-specific events that we've defined in our configuration file (those we would receive as the events occur). We'll call this script catlog.[7]

```
#!/bin/sh
#catlog - This script will tail a log file from an offset point to the end and
then set a new offset point.
#Read file name
fullfile=$1

#change all "/" to "_" for offset file
file=`echo $fullfile | sed 's/\//_/g'`
```

```
#get offset if one exists
if [ -f /tmp/offset.$file ]; then
        offset=`cat /tmp/offset.$file`
else
        offset=0
fi

#get linecount of log file
newoffset=`wc -l $1 | awk '{print $1}'`

#test to see if the file shrunk
if [ $offset -gt $newoffset ]; then
        #If it shrunk read the whole file
        offset=0
else
        #increment offset to avoid the
        #one duplicate line from tail
        offset=$[$offset+1]
fi

#write our new offset
echo $newoffset > /tmp/offset.$file
#tail the file starting from the offset value
tail +$offset $fullfile
```

We can call *catlog* along with Swatch from cron to send us timed reports. Whether we run it nightly or hourly, *catlog* will pass to Swatch only the offset section of the log file called. This relieves Swatch of having to parse through sections of the log file that have been scanned in the previous period. Consequently, the email that we receive consists only of the logged events of the past time period. We would pass catlog to Swatch in this manner:

```
swatch -c ./our_swatch_config_file -p './catlog /var/log/messages' | mail
mshaul@kvw.com
```

Of course we could use various configuration files and pass in a variety of different logs.

Information logging is of utmost importance in a properly secure setting. However, logs by themselves offer the admin little if the files aren't being reviewed. The admin himself (herself) must recognize log files as tools, and must understand the importance of learning to use them effectively by scanning through them regularly. Unfortunately, the commonly overworked admin often only uses the logs in a reactionary way because of a lack of time and because of the sheer size of the logs. Scanning through megabytes of text each day, most of which looks mind-numbingly similar, is not conducive to the exacting processing that a secure machine needs. Fortunately, by using Swatch, the admin can automate log parsing and begin to use the logs in a proactive manner, thereby

reducing lead time, or the headstart, as one of the bad guys' best advantages. Remember that although we may have in place the most advanced firewall devices and IDS software, due to the rate at which new vulnerabilities are discovered, if we're not regularly monitoring our logs, we ourselves will have become the tenuous weak-link in the security fence.

List of References:

1. Garfinkel, Simson and Spafford Gene. Practical Unix and Internet Security, 2nd . Sebastopol, CA: O'Reilly & Associates,Inc. 1996. 289
2. Morton, Matt. Logging and critical logs files: the Decision to Effectively and Proactively Manage System logging and Log Files 9 December 2000. http://www.sans.org/infosecFAQ/securitybasics/logging.htm. (5 May 2001)
3. Garfinkel, Simson and Spafford Gene. Practical Unix and Internet Security, 2nd Sebastopol, CA: O'Reilly & Associates,Inc. 1996. 325
4. Hansen, Stephen E. and Atkins Todd. Centralized System Monitoring With Swatch. 30 July 1992. http://www.stanford.edu/~atkins/swatch/lisa93.html (13 April 2001).
5. Spitzner, Lance. Watching Your Logs. 19 July 2000. http://www.enteract.com/~lspitz/swatch.html (13 April 2001)
6. Chapman, D. Brent and Zwicky, Elizabeth D.. Building Internet Firewalls, 2nd . Sebastopol, CA: O'Reilly & Associates,Inc. 402-403
7. Akin, Thomas. Unix Security class notes. Continuing Education, Kennesaw State University. 1999. Day 3.