

==Phrack Inc.==

Volume 0x0b, Issue 0x3f, Phile #0x03 of 0x14

```
|-----[ L I N E N O I S E ]-----|
|-----|
|-----[ phrack staff ]-----|
```

...all that does not fit anywhere else but which is worth being mentioned in our holy magazine.... enjoy linoise.

```
0x03-1 Analysing suspicious binary files           by Boris Loza
0x03-2 TCP Timestamp to count hosts behind NAT     by Elie aka Lupin
0x03-3 Elliptic Curve Cryptography                 by f86c9203
```

```
|-----[ 0x03-1 ]-----|
|-----Analyzing Suspicious Binary Files and Processes-----|
|-----|
|-----By Boris Loza, PhD-----|
|-----bloza@tegosystemonline.com-----|
|-----|
```

1. Introduction
2. Analyzing a 'strange' binary file
3. Analyzing a 'strange' process
4. Security Forensics using DTrace
5. Conclusion

--[ Introduction

The art of security forensics requires lots of patience, creativity and observation. You may not always be successful in your endeavours but constantly 'sharpening' your skills by hands-on practicing, learning a couple more things here and there in advance will definitely help.

In this article I'd like to share my personal experience in analyzing suspicious binary files and processes that you may find on the system. We will use only standard, out of the box, UNIX utilities. The output for all the examples in the article is provided for Solaris OS.

--[ Analyzing a 'strange' binary file

During your investigation you may encounter some executable (binary) files whose purpose in your system you don't understand. When you try to read this file it displays 'garbage'. You cannot recognize this file by name and you are not sure if you saw it before.

Unfortunately, you cannot read the binary file with more, cat, pg, vi or other utilities that you normally use for text files. You will need other tools. In order to read such files, I use the following tools: strings, file, ldd, adb, and others.

Let's assume, for example, that we found a file called cr1 in the /etc directory. The first command to run on this file is strings(1). This will show all printable strings in the object or binary file:

```
$ strings cr1 | more
```

```

%s %s %s%s%s -> %s%s%s (%.*s)
Version: 2.3
Usage: dsniff [-cdmn] [-i interface] [-s snaplen] [-f services]
          [-t trigger[,...]] [-r|-w savefile] [expression]
...
/usr/local/lib/dsniff.magic
/usr/local/lib/dsniff.services
...

```

The output is very long, so some of it has been omitted. But you can see that it shows that this is actually a dsniff tool masquerading as crl.

Sometimes you may not be so lucky in finding the name of the program, version, and usage inside the file. If you still don't know what this file can do, try to run strings with the 'a' flag, or just '-'. With these options, strings will look everywhere in the file for strings. If this flag is omitted, strings only looks in the initialized data space of the object file:

```
$ strings crl | more
```

Try to compare this against the output from known binaries to get an idea of what the program might be.

Alternatively, you can use the nm(1) command to print a name list of an object file:

```
$ /usr/ccs/bin/nm -p crl | more
```

crl:

[Index]	Value	Size	Type	Bind	Other	Shndx	Name
[180]	0	0	FILE	LOCL	0	ABS	decode_smtp.c
[2198]	160348	320	FUNC	GLOB	0	9	decode_sniffer

Note that the output of this command may contain thousands of lines, depending on the size of the object file. You can run nm through pipe to more or pg, or redirect the output to the file for further analysis.

To check the runtime linker symbol table - calls of shared library routines, use nm with the '-Du' options, where -D displays the symbol table used by ld.so.1 and is present even in stripped dynamic executables, and -u prints a long listing for each undefined symbol.

You can also dump selected parts of any binary file with the dump(1) or elfdump(1) utilities. The following command will dump the strings table of crl binary:

```
$ /usr/ccs/bin/dump -c ./crl | more
```

You may use the following options to dump various parts of the file:

```

-c      Dump the string table(s).
-C      Dump decoded C++ symbol table names.
-D      Dump debugging information.
-f      Dump each file header.
-h      Dump the section headers.

```

```
-l      Dump line number information.
-L      Dump dynamic linking information and static shared library
        information, if available.
-o      Dump each program execution header.
-r      Dump relocation information.
-s      Dump section contents in hexadecimal.
-t      Dump symbol table entries.
```

Note: To display internal version information contained within an ELF file, use the `pvs(1)` utility.

If you are still not sure what the file is, run the command `file(1)`:

```
$ file cr1
cr1:      ELF 32-bit MSB executable SPARC32PLUS Version 1, V8+
Required, UltraSPARC1 Extensions Required, dynamically linked, not
stripped
```

Based on this output, we can tell that this is an executable file for SPARC that requires the availability of libraries loaded by the OS (dynamically linked). This file also is not stripped, which means that the symbol tables were not removed from the compiled binary. This will help us a lot when we do further analysis.

Note: To strip the symbols, do `strip <my_file>`.

The `file` command could also tell us that the binary file is statically linked, with debug output or stripped.

Statically linked means that all functions are included in the binary, but results in a larger executable. Debug output - includes debugging symbols, like variable names, functions, internal symbols, source line numbers, and source file information. If the file is stripped, its size is much smaller.

The `file` command identifies the type of a file using, among other tests, a test for whether the file begins with a certain magic number (see the `/etc/magic` file). A magic number is a numeric or string constant that indicates the file type. See `magic(4)` for an explanation of the format of `/etc/magic`.

If you still don't know what this file is used for, try to guess this by taking a look at which shared libraries are needed by the binary using `ldd(1)` command:

```
$ ldd cr1
...
libsocket.so.1 =>      /usr/lib/libsocket.so.1
librpcsvc.so.1 =>      /usr/lib/librpcsvc.so.1
...
```

This output tells us that this application requires network share libraries (`libsocket.so.1` and `librpcsvc.so.1`).

The `adb(1)` debugger can also be very useful. For example, the following output shows step-by-step execution of the binary in question:

```
# adb cr1
```

```
:s
adb: target stopped at:
ld.so.1`_rt_boot:      ba,a      +0xc
<ld.so.1`_rt_boot+0xc>
,5:s
adb: target stopped at:
ld.so.1`_rt_boot+0x58: st          %l1, [%o0 + 8]
```

You can also analyze the file, or run it and see how it actually works. But be careful when you run an application because you don't know yet what to expect. For example:

```
# adb crl
:r
Using device /dev/hme0 (promiscuous mode)
192.168.2.119 -> web      TCP D=22 S=1111 Ack=2013255208
Seq=1407308568 Len=0 Win=17520
      web -> 192.168.2.119 TCP D=1111 S=22 Push Ack=1407308568
```

We can see that this program is a sniffer. See `adb(1)` for more information of how to use the debugger.

If you decide to run a program anyway, you can use `truss(1)`. The `truss` command allows you to run a program while outputting system calls and signals.

Note: `truss` produces lots of output. Redirect the output to the file:

```
$ truss -f -o cr.out ./crl
listening on hme0
^C
$
```

Now you can easily examine the output file `cr.out`.

As you can see, many tools and techniques can be used to analyze a strange file. Not all files are easy to analyze. If a file is a statically linked stripped binary, it would be much more difficult to find what a file (program) is up to. If you cannot tell anything about a file using simple tools like `strings` and `ldd`, try to debug it and use `truss`. Experience using and analyzing the output of these tools, together with a good deal of patience, will reward you with success.

#### --[ Analyzing a 'strange' process

What do you do if you find a process that is running on your system, but you don't know what it is doing? Yes, in UNIX everything is a file, even a process! There may be situations in which the application runs on the system but a file is deleted. In this situation the process will still run because a link to the process exists in the `/proc/[PID]/object/a.out` directory, but you may not find the process by its name running the `find(1)` command.

For example, let's assume that we are going to investigate the process ID 22889 from the suspicious `srg` application that we found running on our system:

```
# ps -ef | more
UID  PID  PPID  C   STIME TTY      TIME CMD
...
root 22889 16318 0 10:09:25 pts/1    0:00 ./srg
...
```

Sometimes it is as easy as running the strings(1) command against the /proc/[PID]/object/a.out to identify the process.

```
# strings /proc/22889/object/a.out | more
...
TTY-Watcher version %s
Usage: %s [-c]
-c  turns on curses interface
NOTE: Running without root privileges will only allow you to monitor
yourself.
...
```

We can see that this command is a TTY-Watcher application that can see all keystrokes from any terminal on the system.

Suppose we were not able to use strings to identify what this process is doing. We can examine the process using other tools.

You may want to suspend the process until you will figure out what it is. For example, run kill -STOP 22889 as root. Check the results. We will look for 'T' which indicates the process that was stopped:

```
# /usr/ucb/ps | grep T
root      22889  0.0  0.7 3784 1720 pts/1    T 10:09:25  0:00 ./srg
```

Resume the process if necessary with kill -CONT <PID>  
To further analyze the process, we will create a \core dump\ of variables and stack of the process:

```
# gcore 22889
gcore: core.22889 dumped
```

Here, 22889 is the process ID (PID). Examine results of the core.22889 with strings:

```
# strings core.22889 | more
...
TTY-Watcher version %s
Usage: %s [-c]
-c  turns on curses interface
NOTE: Running without root privileges will only allow you to monitor
yourself.
...
```

You may also use coreadm(1M) to analyze the core.22889 file. The coreadm tool provides an interface for managing the parameters that affect core file creation. The coreadm command modifies the /etc/coreadm.conf file. This file is read at boot time and sets the global parameters for core dump creation.

First, let's set our core filenames to be of the form core.<PROC NAME>.<PID>.

We'll do this only for all programs we execute in this shell (the \$\$ notation equates to the PID of our current shell):

```
$ coreadm -p core.%f.%p $$
```

The %f indicates that the program name will be included, and the %p indicates that the PID will be appended to the core filename.

You may also use adb to analyze the process. If you don't have the object file, use the /proc/[PID]/object/a.out. You can use a core file for the process dumped by gcore or specify a '-' as a core file. If a dash (-) is specified for the core file, adb will use the system memory to execute the object file. You can actually run the object file under the adb control (it could also be dangerous because you don't know for sure what this application is supposed to do!):

```
# adb /proc/22889/object/a.out -
main:b
:r
breakpoint at:
main:          save    %sp, -0xf8, %sp
...
:s
stopped at:
main+4:        clr     %l0
:s
stopped at:
main+8:        sethi   %hi(0x38400), %o0
$m
? map
...
b11 = ef632f28 e11 = ef6370ac f11 = 2f28 `/usr/lib/libsocket.so.1'
$q
```

We start the session by setting a breakpoint at the beginning of main() and then begin execution of a.out by giving adb the ':r' command to run. Immediately, we stop at main(), where our breakpoint was set. Next, we list the first instruction from the object file. The ':s' command tells adb to step, executing only one assembly instruction at a time.

Note: Consult the book Panic!, by Drake and Brown, for more information on how to use adb to analyze core dumps.

To analyze the running process, use truss:

```
# truss -vall -f -o /tmp/outfile -p 22889
# more /tmp/outfile
```

On other UNIX systems, where available, you may trace a process by using the ltrace or strace commands. To start the trace, type ltrace -p <PID>.

To view the running process environment, you may use the following:

```
# /usr/ucb/ps auxeww 22889
USER      PID %CPU %MEM  SZ  RSS TT      S    START  TIME COMMAND
root      22889 0.0  0.4 1120  896    pts/1  S    14:15:27  0:00 -
sh _=/usr/bin/csh
```

```
MANPATH=/usr/share/man:/usr/local/man HZ=
PATH=/usr/sbin:/usr/bin:/usr/local/bin:/usr/ccs/bin:/usr/local/sbin:
/opt/NSCPcom/ LOGNAME=root SHELL=/bin/ksh HOME=/
LD_LIBRARY_PATH=/usr/openwin/lib:/usr/local/lib TERM=xterm TZ=
```

The /usr/ucb directory contains SunOS/BSD compatibility package commands. The /usr/ucb/ps command displays information about processes. We used the following options (from the man for ps(1B)):

```
-a      Include information about processes owned by others.
-u      Display user-oriented output. This includes fields USER, %CPU, o
        %MEM, SZ, RSS and START as described below.
-x      Include processes with no controlling terminal.
-e      Display the environment as well as the arguments to the command.
-w      Use a wide output format (132 columns rather than 80); if repeated,
        that is, -ww, use arbitrarily wide output. This information is
        used to decide how much of long commands to print.
```

To view the memory address type:

```
# ps -ealf | grep 22889
 F S      UID  PID  PPID  C PRI NI      ADDR      SZ      WCHAN
STIME    TTY      TIME CMD
8 S      root 3401 22889 0 41 20 615a3b40 474 60ba32e6 14:16:49
pts/1    0:00 ./srg
```

To view the memory usage, type:

```
# ps -e -opid,vsz,rss,args
  PID  VSZ  RSS COMMAND
...
22889 3792 1728 ./srg
```

We can see that the ./srg uses 3,792 K of virtual memory, 1,728 of which have been allocated from physical memory.

You can use the /etc/crash(1M) utility to examine the contents of a proc structure of the running process:

```
# /etc/crash
dumpfile = /dev/mem, namelist = /dev/ksyms, outfile = stdout
> p
PROC TABLE SIZE = 3946
SLOT ST  PID  PPID  PGID  SID  UID PRI  NAME      FLAGS
...
 66 s 22889 16318 16337 24130 0 58 srg      load
> p -f 66
PROC TABLE SIZE = 3946
SLOT ST  PID  PPID  PGID  SID  UID PRI  NAME      FLAGS
 66 s 22889 16318 16337 24130 0 58 srg      load

      Session: sid: 24130, ctty: vnode(60b8f3ac) maj( 24) min( 1)
      ...
>
```

After invoking the crash utility, we used the p function to get the process table slot (66, in this case). Then, to dump the proc structure for process

PID 22889, we again used the p utility, with the '-f' flag and the process table slot number.

Like the process structure, the uarea contains supporting data for signals, including an array that defines the disposition for each possible signal. The signal disposition tells the operating system what to do in the event of a signal - ignore it, catch it and invoke a user-defined signal handler, or take the default action. To dump a process's uarea:

```
> u 66
PER PROCESS USER AREA FOR PROCESS 66
PROCESS MISC:
    command: srg, psargs: ./srg
    start: Mon Jun  3 08:56:40 2002
    mem: 6ad, type: exec su-user
    vnode of current directory: 612daf48
...
>
```

The 'u' function takes a process table slot number as an argument. To dump the address space of a process, type:

```
# /usr/proc/bin/pmap -x 22889
```

To obtain a list of process's open files, use the /usr/proc/bin/pfiles command:

```
# /usr/proc/bin/pfiles 22889
```

The command lists the process name and PID for the process' open files. Note that various bits of information are provided on each open file, including the file type, file flags, mode bits, and size.

If you cannot find a binary file and the process is on the memory only, you can still use methods described for analyzing suspicious binary files above against the process's object file. For example:

```
# /usr/ccs/bin/nm a.out | more
a.out:
```

[Index]	Value	Size	Type	Bind	Other	Shndx	Name
...							
[636]	232688	4	OBJT	GLOB	0	17	Master_utmp
[284]	234864	20	OBJT	GLOB	0	17	Mouse_status

You may also use mdb(1) - a modular debugger to analyze the process:

```
# mdb -p 22889
Loading modules: [ ld.so.1 libc.so.1 libnvpair.so.1 libuutil.so.1 ]
> ::objects
    BASE    LIMIT    SIZE NAME
    10000    62000    52000 ./srg
ff3b0000 ff3dc000    2c000 /lib/ld.so.1
ff370000 ff37c000     c000 /lib/libsocket.so.1
ff280000 ff312000    92000 /lib/libnsl.so.1
```

--[ Security Forensics using DTrace

Solaris 10 has introduced a new tool for Dynamic Tracing in the OS environment - dtrace. This is a very powerful tool that allows system administrators to observe and debug the OS behaviour or even to dynamically modify the kernel. Dtrace has its own C/C++ like programming language called 'D language' and comes with many different options that I am not going to discuss here. Consult dtrace(1M) man pages and <http://docs.sun.com/app/docs/doc/817-6223> for more information.

Although this tool has been designed primarily for developers and administrators, I will explain how one can use dtrace for analyzing suspicious files and process.

We will work on a case study, as follows. For example, let's assume that we are going to investigate the process ID 968 from the suspicious srg application that we found running on our system.

By typing the following at the command-line, you will list all files that this particular process opens at the time of our monitoring. Let it run for a while and terminate with Control-C:

```
# dtrace -n syscall::open:entry'/pid == 968/  
{ printf("%s%s",execname,copyinstr(arg0)); }'
```

```
dtrace: description 'syscall::open*:entry' matched 2 probes
```

```
^C
```

CPU	ID	FUNCTION:NAME
0	14	open:entry srg /var/ld/ld.config
0	14	open:entry srg /lib/libdhcputil.so.1
0	14	open:entry srg /lib/libsocket.so.1
0	14	open:entry srg /lib/libnsl.so.1

D language comes with its own terminology, which I will try to address here briefly.

The whole 'syscall::open:entry' construction is called a 'probe' and defines a location or activity to which dtrace binds a request to perform a set of 'actions'. The 'syscall' element of the probe is called a 'provider' and, in our case, permits to enable probes on 'entry' (start) to any 'open' Solaris system call ('open' system call instructs the kernel to open a file for reading or writing).

The so-called 'predicate' - /pid == 968/ uses the predefined dtrace variable 'pid', which always evaluates to the process ID associated with the thread that fired the corresponding probe.

The 'execname' and 'copyinstr(arg0)' are called 'actions' and define the name of the current process executable file and convert the first integer argument of the system call (arg0) into a string format respectively. The printf's action uses the same syntax as in C language and serves for the same purpose - to format the output.

Each D program consists of a series of 'clauses', each clause describing one or more probes to enable, and an optional set of actions to perform when the probe fires. The actions are listed as a series of statements enclosed in curly braces { } following the probe name. Each statement ends with a semicolon (;).

You may want to read the Introduction from Solaris Tracing Guide (<http://docs.sun.com/app/docs/doc/817-6223>) for more options and to understand the syntax.

Note: As the name suggests, the dtrace (Dynamic Trace) utility will show you the information about a changing process - in dynamic. That is, if the process is idle (doesn't do any system calls or opens new files), you won't be able to get any information. To analyze the process, either restart it or use methods described in the previous two sections of this paper.

Second, we will use the following command-line construction to list all system calls for 'srg'. Let it run for a while and terminate by Control-C:

```
# dtrace -n 'syscall:::entry /execname == "srg"/ { @num[probefunc] =
count(); }'
dtrace: description 'syscall:::entry ' matched 226 probes
^C
  pollsys                                1
  getrlimit                               1
  connect                                 1
  setsockopt                              1
  ...
```

You may recognize some of the building elements of this small D program. In addition, this clause defines an array named 'num' and assigns the appropriate member 'probefunc' (executed system call's function) the number of times these particular functions have been called (count()).

Using dtrace we can easily emulate all utilities we have used in the previous sections to analyze suspicious binary files and processes. But dtrace is much more powerful tool and may provide one with more functionality: for example, you can dynamically monitor the stack of the process in question:

```
# dtrace -n 'syscall:::entry/execname == "srg"/{ustack()}'
0    286          lwp_sigmask:entry
          libc.so.1`__systemcall6+0x20
          libc.so.1`pthread_sigmask+0x1b4
          libc.so.1`sigprocmask+0x20
          srg`srg_alarm+0x134
          srg`scan+0x400
          srg`net_read+0xc4
          srg`main+0xabc
          srg`_start+0x108
```

Based on all our investigation (see the list of opened files, syscalls, and the stack examination above), we may positively conclude that srg is a network based application. Does it write to the network? Let's check it by constructing the following clause:

```
# dtrace -n 'mib:ip:::/execname == "srg"/{@[execname]=count()}'
dtrace: description 'mib:ip:::' matched 412 probes
dtrace: aggregation size lowered to 2m
^C
  srg                                520
```

It does. We used 'mib' provider to find out if our application transmits to the network.

Could it be just a sniffer or a netcat-liked application that is bounded to a specific port? Let's run dtrace in the truss(1) like fashion to answer this question (inspired by Brendan Gregg's dtruss utility ):

```
#!/usr/bin/sh
#
dtrace='

inline string cmd_name = "'$1'";
/*
** Save syscall entry info
*/
syscall::entry
/execname == cmd_name/
{
    /* set start details */
    self->start = timestamp;
    self->arg0 = arg0;
    self->arg1 = arg1;
    self->arg2 = arg2;
}

/* Print data */
syscall::write:return,
syscall::pwrite:return,
syscall::*read*:return
/self->start/
{
    printf("%s(0x%X, \"%S\", 0x%X)\t\t = %d\n", probefunc, self->arg0,
        stringof(copyin(self->arg1, self->arg2)), self->arg2, (int)arg0);

    self->arg0 = arg0;
    self->arg1 = arg1;
    self->arg2 = arg2;
}
,
# Run dtrace
    /usr/sbin/dtrace -x evaltime=exec -n "$dtrace" >&2
```

Save it as truss.d, change the permissions to executable and run:

```
# ./truss.d srg
0      13                               write:return write(0x1, "          sol10 -
> 192.168.2.119 TCP D=3138 S=22 Ack=713701289 Seq=3755926338 Len=0
Win=49640\n8741 Len=52 Win=16792\n\0", 0x5B)          = 91
0      13                               0      13
write:return write(0x1, "192.168.2.111 -> 192.168.2.1  UDP D=1900
S=21405 LEN=140\n\0", 0x39)          = 57
^C
```

Looks like a sniffer to me, with probably some remote logging (remember the network transmission by ./srg discovered by the 'mib' provider above!).

You can actually write pretty sophisticated programs for dtrace using D language.

Take a look at /usr/demo/dtrace for some examples.

You may also use dtrace for other forensic activities. Below is an example of more complex script that allows monitoring of who fires the suspicious application and starts recording of all the files opened by the process:

```
#!/usr/bin/sh

command=$1

/usr/sbin/dtrace -n '

inline string COMMAND = '$command';

#pragma D option quiet

/*
** Print header
*/
dtrace:::BEGIN
{
    /* print headers */
    printf("%-20s %5s %5s %5s %s\n", "START_TIME", "UID", "PID", "PPID", "ARGS");
}

/*
** Print exec event
*/
syscall::exec:return, syscall::exece:return
/(COMMAND == execname)/
{
    /* print data */
    printf("%-20Y %5d %5d %5d %s\n", walltimestamp, uid, pid, ppid,
        stringof(curpsinfo->pr_psargs));
    s_pid = pid;
}

/*
** Print open files
*/
syscall::open*:entry
/pid == s_pid/
{
    printf("%s\n", copyinstr(arg0));
}
',
```

Save this script as wait.d, change the permissions to executable 'chmod +x wait.d' and run:

```
# ./wait.d srg
START_TIME          UID   PID  PPID ARGS
2005 May 16 19:51:20  100  1582  1458 ./srg
```

```
/var/ld/ld.config
```

```
/lib/libnsl.so.1
/lib/libsocket.so.1
/lib/libresolv.so.2
...
^C
```

Once the srg is started you will see the output.

However, the real power of dtrace comes from the fact that you can do things with it that won't be possible without writing a comprehensive C program. For example, the shellsnoop application written by Brendan Gregg (<http://users.tpg.com.au/adsl4yb/DTrace/shellsnoop>) allows you to use dtrace at the capacity of ttywatcher!

It is not possible to show all capabilities of dtrace in such a small presentation of this amazing utility. Dtrace is a very powerful as well a complex tool with virtually endless capabilities. Although Sun insists that you don't have to have a 'deep understanding of the kernel for DTrace to be useful', the knowledge of Solaris internals is a real asset. Taking a look at the include files in /usr/include/sys/ directory may help you to write complex D scripts and give you more of an understanding of how Solaris 10 is implemented.

--[ Conclusion

Be creative and observant. Apply all your knowledge and experience for analyzing suspicious binary files and processes. Also, be patient and have a sense of humour!

f86c92039c992d2d2bd2b85c8807ac2f7af57c5c

|=[ EOF ]=-----=|