CENTER FOR TRUSTWORTHY SCIENTIFIC CYBERINFRASTRUCTURE
The NSF Cybersecurity Center of Excellence

perfSONAR-CTSC Code Review

January 11, 2016

For Public Distribution

Randy Heiland, Andrew Adams, Elisa Heymann

About CTSC

The mission of the Center for Trustworthy Scientific Cyberinfrastructure (CTSC, trustedci.org) is to improve the cybersecurity of NSF science and engineering projects, while allowing those projects to focus on their science endeavors. This mission is accomplished through one-on-one engagements with projects to solve their specific problems, broad education, outreach and training to raise the practice-of-security across the community, and looking for opportunities for improvement to bring in research to raise the state-of-practice.

Acknowledgments

CTSC's engagements are inherently collaborative; the authors would like to thank the perfSONAR team for the collaborative effort that made this document possible.

This document is a product of the Center for Trustworthy Scientific Cyberinfrastructure (CTSC). CTSC is supported by the National Science Foundation under Grant Numbers OCI-1234408 and ACI-1547272. For more information about the Center for Trustworthy Scientific Cyberinfrastructure please visit: http://trustedci.org/. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect the views of the National Science Foundation.

Using & Citing this Work

Cite this work using the following information:
R. Heiland, A. Adams, and E. Heymann, "perfSONAR-CTSC Code Review," Center for Trustworthy Scientific Cyberinfrastructure, trustedci.org, Jan 2016. Available: http://hdl.handle.net/2022/20596

This work is available on the web at the following URL:
http://trustedci.org/perfsonar

Table of Contents

Executive Summary

perfSONAR ("Performance focused Service Oriented Network monitoring ARchitecture") is an infrastructure for monitoring network performance. The perfSONAR software toolkit is deployed around the world, primarily at government labs and universities, to help monitor and provide network reliability information across multiple domains. Some of the virtual organizations deploying perfSONAR include ESnet, GÉANT, and Internet2.

CTSC and perfSONAR conducted an engagement in which CTSC performed a code review of perfSONAR's Bandwidth Test Controller (BWCTL). BWCTL is essentially a daemon and framework for scheduling and executing non-overlapping performance measurement tests between sets of participating hosts (endpoints). The code review consisted of two parts: (1) a First Principles Vulnerability Assessment (FPVA) that involved a manual inspection and analysis of the code, resulting in detailed architecture and resources diagrams and (manual) detection of potential vulnerabilities, and (2) an automated/programmatic static source code analysis using the Software Assurance Marketplace (SWAMP) online service.

Overall, the review of the existing code was quite positive. While there were concerns with the use of C string commands (str*()), the code takes sufficient care to minimize vulnerabilities. BWCTL uses exec*() function calls and spawns processes via fork() and therefore we recommend sanitizing the environment at the very beginning to avoid sabotage of environment variables, potentially resulting in vulnerabilities during execution. The static analysis did not flag any bugs as security errors; however, there were several classified as memory errors that we recommend be fixed. Looking to the future, CTSC suggests that the perfSONAR team consider adopting SELinux with a bwctl targeted policy module for its endpoints. Finally, because BWCTL relies on the Network Time Protocol (NTP), we recommend following the progress of and eventually adopting NTPsec (http://www.ntpsec.org/) over NTP classic (http://www.ntp.org/). Currently, NTPsec is in a public beta release; we recommend waiting for the stable release.

1 First Principles Vulnerability Assessment

This section provides results from a First Principles Vulnerability Assessment (FPVA) [1] (with a focused code analysis) for the perfSONAR Bandwidth Test Controller (BWCTL[1]) code. We would estimate our effort for the FPVA task to be about two person-months.

[1] https://github.com/perfsonar/bwctl

1.1 Architectural analysis

The Bandwidth Test Controller (BWCTL) system is a core part of the perfSONAR project. The major structural components of the BWCTL system (Figure 1) include the **endpoint** server hosts that perform the bandwidth tests and the **client** hosts that make the requests and obtain the results. Each endpoint runs a server, the `bwctld` daemon, that forks off a *resource broker* process (arrow 1). The basic use case for a bandwidth test is that a client, `bwctl`, initiates a test between two endpoints (arrow 2). This causes the resource broker to fork a *request process* (arrow 3) that will determine whether or not the request is valid. If it is valid, the request process will request from the resource broker (bidirectional vertical arrow) the resources and time period requested from the client. Assuming those can be met, the resource broker grants the request. At this point, the request process forks a *peer process* (arrow 4) that will verify the time offset to the other endpoint and initialize the socket used to communicate the results of the test (arrow 5). Assuming the two endpoints are able to communicate and both know the correct time, the peer process will fork a *test process* (arrow 6) that will, at the test's start time, execute the requested test program (with any parameters) (arrow 7). The test results will be communicated back to the client (arrow 8).

BWCTL relies on the Network Time Protocol (NTP) to synchronize the timing of tests between endpoints. All code for BWCTL is written in C (about 25K lines of code spread over about 80 files) and makes extensive use of Unix network programming, including the creation of new processes (via fork()), sockets, and signals. It does not, however, use threads. The attack surface includes the interfaces that are available to users for providing input to the system. These include the client's command line arguments and the server's command line arguments and configuration file parameters. Figure 1 depicts an architecture diagram for BWCTL, showing a client establishing a test between two servers. (Note: the bwctld daemon is run as "bwctl" which is only in the group "bwctl" (groups bwctl)). The following output from the 'ps' command reveals the user IDs associated with the client (bwctl), the server/daemon (bwctld) and its forked processes:

```
[root@gw44 ~]# ps -ef|grep bwctl
bwctl    10466     1  0 Oct23 ?        00:00:03 /usr/bin/bwctld -c
/etc/bwctld -R /var/run
heiland  12970 12946  0 09:29 pts/1    00:00:00 /usr/bin/bwctl -T
iperf3 -f m -t 10 -i 1 -c llnl-pt1.es.net -v
bwctl    12971 10466  0 09:29 ?        00:00:00 /usr/bin/bwctld -c
/etc/bwctld -R /var/run
bwctl    12974 12971  0 09:29 ?        00:00:00 /usr/bin/bwctld -c
/etc/bwctld -R /var/run
```

```
bwctl     12975 12974   6 09:29 ?         00:00:02 iperf3 -c
198.129.254.106 -B 149.165.228.236 -f m -p 5581 -i 1.000000 -V -Z -t
10
```

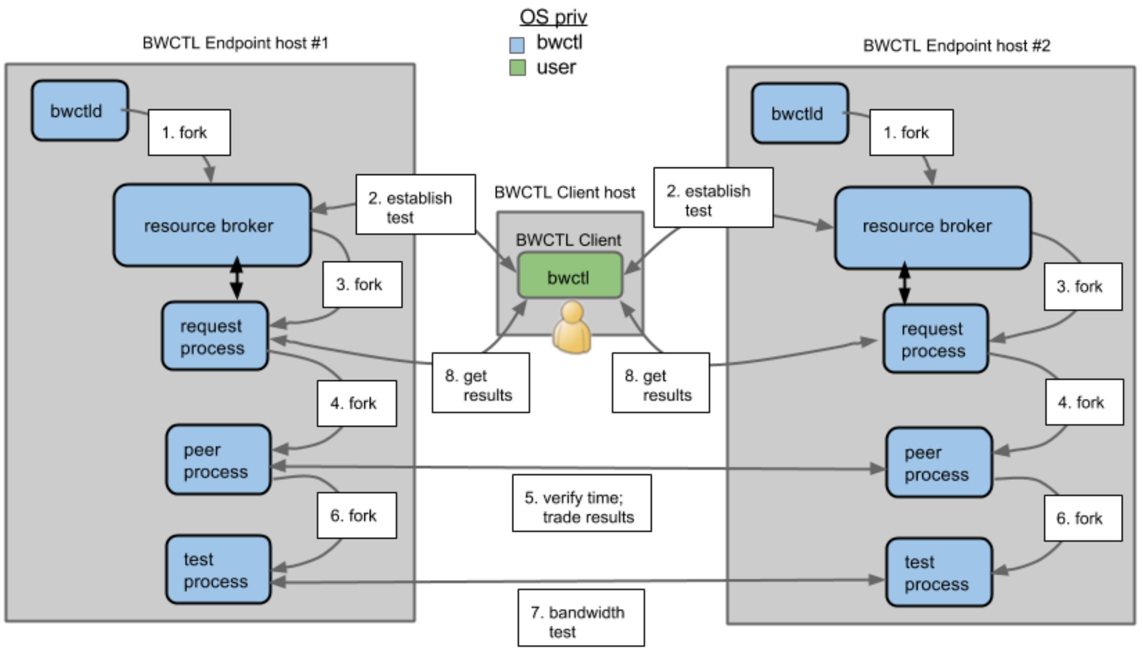

Figure 1. perfSONAR BWCTL Architecture Diagram: processes and flow of execution

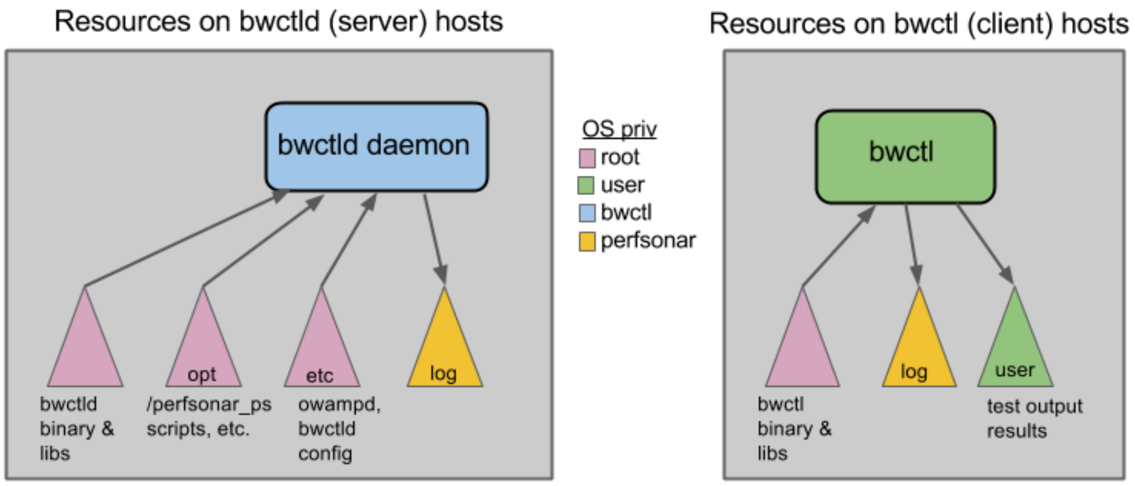

Figure 2. perfSONAR BWCTL Resources

1.2 Resource identification

Figure 2 depicts the key resources used in BWCTL: the endpoint server hosts (running the bwctld daemon), the client hosts, configuration files, log files (on both server and client hosts), and (optionally) output files of test results at the client. Other resources (not depicted in Fig. 2) will be CPU cycles and network bandwidth used by the hosts. Ideally, the hosts would be standalone computers with no sensitive information stored on them; however, this decision is ultimately left to the administrators of those systems. There are no databases associated with the BWCTL system.

1.3 Trust and privilege analysis

Each endpoint host can have its own degree of trust: a function of the physical security of the facility and the software security of its operating systems, libraries, and BWCTL dependencies (including the test utilities). Associated with trust is the privilege level at which each executable component runs. The privilege levels control the extent of access for each component and, in the case of exploitation, the extent of damage than could occur. The fact that multiple processes are forked within the bwctld server means that certain privileges are being delegated to those processes.

1.4 Component evaluation

In this component of the FPVA process, we have attempted to manually examine key pieces of the BWCTL code base. In this case, we examined the C code for the server (bwctld.c) and client (bwctl.c), as well as supporting code (/bwlib). Some potential vulnerabilities include buffer overflow (due to the use of strcat(), etc) and the use of the *exec** system function to execute user-supplied scripts. However, in looking at the source code, all instances of the strcpy(3) and strcat(3) routines either (i) use internal arguments, e.g., #DEFINE values, (ii) are preceded by strlen(3) checks on the untainted variables to verify that the copy will succeed, or (iii) are used with strdup(3) returned pointers, i.e., strdup(3) creates a sufficiently sized buffer to hold the external variable. Hence, the team sees no problem in the BWCTL source code with the use of non-buf-length-checking string routines. Similarly, BWCTL uses the execve(2) family of calls to execute external programs, thereby averting many vulnerabilities associated with the use of system(3). Note, the code does *not* appear to have included checks that each *command* variable passed to execlp(3) or execvp(3) is initiated with a "/" value, thus avoiding the potential threat in relying on the PATH environment variable (see [2]). Moreover, the code does not check to verify that the user is incapable of modifying the file pointed to by the *command* variables (see [3]). However, we believe the risk of both threats is minor.

Ideally, to reduce the risk the perfSONAR project incurs due to the use of a compromised or misused BWCTL, we recommend a mandatory access control mechanism like SELinux[2]. Since perfSONAR nodes are distributed on CentOS 6, SELinux is available, and the more tractable option of enabling it in targeted mode exists[3]. Unfortunately, as of this engagement, CentOS 6's default targeted policy module does not contain a domain for bwctl. If SELinux is enabled in targeted mode, bwctl will be assigned to the unconfined_t domain, and executed with standard Unix permissions, unconstrained by SELinux. Thus, the preferred solution is to create a targeted policy module that can be loaded into a SELinux enabled kernel which is capable of containing bwctl. The steps to develop a policy module from scratch are nontrivial. In short, (i) a new policy for the bwctl domain must be created and loaded, (ii) the bwctl executable must be assigned to the new domain, (iii) the application must be executed in permissive mode (this will log all audit messages), and (iv) audit2allow is run over the audit messages to build the actual targeted policy (see [4] and [5] for information regarding these steps).

2 Static Code Analysis (via SWAMP)

In addition to the FPVA analysis described above, CTSC also performed a static code analysis of the BWCTL code using the Software Assurance Marketplace (SWAMP[4] [5]): a no-cost, high-performance, centralized cloud computing platform that includes both open-source and commercial software security testing tools. SWAMP also offers options for viewing results from an analysis.

Although CTSC had some experience with SWAMP, using it to analyze the BWCTL code presented a bit of a challenge. (We are happy to share our experience with perfSONAR staff if that would be useful). Figure 3 shows results from SWAMP, running the Clang Static Analyzer tool on a snapshot of the BWCTL code. It found 28 "bugs" in the code, however, none were found in the Security Checker classification (http://clang-analyzer.llvm.org/available_checks.html#security_checkers). In spite of this, CTSC still suggests following the recommendations in the previous section.

Total	API	Dead store	Logic error	Memory Error	Security	Unix API
28	2	13	3	10	0	0

Category	File	Line	Message
API	bwctl/I2util/pfstore/pfstore.c	279	Null pointer passed as an argument to a 'nonnull' parameter

[2] http://selinuxproject.org/page/Main_Page
[3] https://www.centos.org/docs/5/html/Deployment_Guide-en-US/sec-sel-policy-targeted-oview.html
[4] https://continuousassurance.org/
[5] https://continuousassurance.org/swamp/SWAMP-WP002-Framework.pdf

API	bwctl/I2util/I2util/conf.c	641	Null pointer passed as an argument to a 'nonnull' parameter
Dead store	bwctl/bwlib/tools.c	133	Value stored to 'n' is never read
Dead store	bwctl/I2util/I2util/ErrLogSyslog.c	406	Value stored to 'size' is never read
Dead store	bwctl/bwlib/endpoint.c	1549	Value stored to 'aval' is never read
Dead store	bwctl/I2util/I2util/hmac-sha1.c	157	Value stored to 'keylen' is never read
Dead store	bwctl/bwctld/policy.c	1400	Value stored to 'ret' is never read
Dead store	bwctl/bwctl/bwctl.c	3018	Value stored to 'tid' is never read
Dead store	bwctl/bwlib/paris-traceroute.c	141	Value stored to 'local_side' is never read
Dead store	bwctl/bwctld/bwctld.c	2311	Value stored to 'argv' is never read
Dead store	bwctl/I2util/I2util/hmac-sha1.c	161	Value stored to 'keylen' is never read
Dead store	bwctl/bwlib/paris-traceroute.c	137	Value stored to 'local_side' is never read
Dead store	bwctl/bwlib/protocol.c	920	Value stored to 'omit_available' is never read
Dead store	bwctl/bwctld/policy.c	1393	Value stored to 'ret' is never read
Dead store	bwctl/bwlib/protocol.c	1278	Value stored to 'omit_available' is never read
Logic error	bwctl/bwlib/endpoint.c	1541	The left operand of '>=' is a garbage value
Logic error	bwctl/bwlib/endpoint.c	1239	Access to field 'sockfd' results in a dereference of a null pointer (loaded from field 'rcntrl')
Logic error	bwctl/I2util/I2util/conf.c	956	Division by zero
Memory Error	bwctl/bwctld/policy.c	467	Potential leak of memory pointed to by 'tnode.limits'
Memory Error	bwctl/bwlib/context.c	216	Use of memory after it is freed
Memory Error	bwctl/bwctld/policy.c	889	Potential leak of memory pointed to by 'policy'
Memory Error	bwctl/bwctld/policy.c	467	Potential leak of memory pointed to by 'tnode.nodename'
Memory Error	bwctl/bwctld/bwctld.c	2027	Potential leak of memory pointed to by 'new_posthook'
Memory	bwctl/bwctl/bwctl.c	3711	Potential leak of memory pointed to

Error			by 'scheduled_times_schedule'
Memory Error	bwctl/I2util/I2util/hmac-sha1.c	110	Potential leak of memory pointed to by 'hmac'
Memory Error	bwctl/I2util/I2util/random.c	79	Potential leak of memory pointed to by 'rand_src'
Memory Error	bwctl/bwctld/policy.c	467	Potential leak of memory pointed to by 'node'
Memory Error	bwctl/bwctld/policy.c	479	Potential leak of memory pointed to by 'tnode.used'

Figure 3. Output results of an initial SWAMP run on BWCTL code

References

[1] James A. Kupsch, Barton P. Miller, Eduardo César, and Elisa Heymann, "First Principles Vulnerability Assessment", *2010 ACM Cloud Computing Security Workshop (CCSW)*, Chicago, IL, October 2010. URL. http://research.cs.wisc.edu/mist/papers/ccsw12sp-kupsch.pdf

[2] CERT: *Sanitize the environment when invoking external programs* URL. https://www.securecoding.cert.org/confluence/display/c/ENV03-C.+Sanitize+the+environment+when+invoking+external+programs

[3] CERT: *Do not call system()* URL. https://www.securecoding.cert.org/confluence/pages/viewpage.action?pageId=2130132

[4] CentOS: *Targeted Policy Overview* URL. https://www.centos.org/docs/5/html/Deployment_Guide-en-US/sec-sel-policy-targeted-oview.html

[5] CentOS: *Writing an SELinux module* URL. http://www.billauer.co.il/selinux-policy-module-howto.html#SECTION00060000000000000000

Appendix

In this Appendix, we take a closer look at the use of BWCTL and highlight some shell commands and tools that might be helpful with the FPVA process.

Accessing/Classifying the Code

To obtain the BWCTL code, we cloned it from its git repository:

```
/tmp$ git clone https://github.com/perfsonar/bwctl.git
Cloning into 'bwctl'...
remote: Counting objects: 5547, done.
remote: Total 5547 (delta 0), reused 0 (delta 0), pack-reused
5547
Receiving objects: 100% (5547/5547), 7.11 MiB | 7.34 MiB/s,
done.
Resolving deltas: 100% (1947/1947), done.
Checking connectivity... done.
```

After bundling in the **I2util** library that contains several utility functions used by BWCTL, we end up with:

```
/tmp$ perl ~/Downloads/cloc-1.64.pl bwctl
     162 text files.
     150 unique files.
      44 files ignored.

http://cloc.sourceforge.net v 1.64  T=1.03 s (115.4 files/s, 45373.9 lines/s)
-------------------------------------------------------------------------------
Language                     files          blank        comment           code
-------------------------------------------------------------------------------
C                               51           4036           9228          21877
C/C++ Header                    29            728           2197           3241
Perl                             7            342            562           1686
HTML                             5            171             43            991
m4                               7             58             11            461
XML                              3             20             28            151
make                            13             57            440            134
Bourne Shell                     3             19             77            132
Bourne Again Shell               1             13             27             79
-------------------------------------------------------------------------------
SUM:                           119           5444          12613          28752
-------------------------------------------------------------------------------
```

Options for running a server on an endpoint

We followed the instructions here (http://docs.perfsonar.net/install_centos.html) to install the perfSONAR Toolkit as rpm bundles on an existing CentOS host (endpoint) and start the `bwctld` daemon. In general, here are the options for running the daemon:

```
$ bwctld -h

Usage: bwctld [options]

Where "options" are:

   -a authmode       Default supported
authmodes:[E]ncrypted,[A]uthenticated,[O]pen
   -c confdir        Configuration directory
   -e facility       syslog facility to log errors
   -f                Allow daemon to run as "root" (folly!)
   -G group          Run as group "group" :-gid also valid
   -h                Print this message and exit
   -R vardir         Location for pid file
   -S nodename:port  Srcaddr to bind to
      -U/-G options only used if run as root
   -U user           Run as user "user" :-uid also valid
   -V                version
   -w                Debugging: busy-wait children after fork to allow
attachment
   -Z                Debugging: Run in foreground

Version: 1.5.5-1
```

Options for running a client

To get "help" running a client, one can execute the following command:

```
$ bwctl -h
bwctl:
usage: bwctl [arguments]

Connection Arguments
-4|--ipv4                        Use IPv4 only
-6|--ipv6                        Use IPv6 only
```

```
-B|--local_address <address>     Use this as a local address for control
connection and tests
-c|--receiver <address>          The host that will act as the receiving side
for a test
-E|--no_endpoint                 Allow tests to occur when the receiver isn't
running bwctl (Default: False)
-o|--flip                        Have the receiver connect to the sender
(Default: False)
-s|--sender <address>            The host that will act as the sending side
for a test

Scheduling Arguments
-a|--allow_ntp_unsync <seconds>  Allow unsynchronized clock - claim good
within offset
-I|--test_interval <seconds>     Time between repeated bwctl tests
-L|--latest_time <seconds>       Latest time into an interval to allow a test
to run
-n|--num_tests <num>             Number of tests to perform (Default: 1)
-R|--randomize <percent>         Randomize the start time within this
percentage of the test's interval (Default: 10%)
--schedule <schedule>            Specify the specific times when a test
should be run (e.g. --schedule 11:00,13:00,15:00)
--streaming                      Request the next test as soon as the current
test finishes

Test Arguments
-b|--bandwidth <bandwidth>       Bandwidth to use for tests (bits/sec KM)
(Default: 1Mb for UDP tests, unlimited for TCP tests)
-D|--dscp <dscp>                 RFC 2474-style DSCP value for TOS byte
-i|--report_interval <seconds>   Tool reporting interval
-l|--buffer_length <bytes>       Length of read/write buffers
-O|--omit <seconds>              Omit time (currently only for iperf3)
-P|--parallel <num>              Number of concurrent connections
-S|--tos <tos>                   Type-Of-Service for outgoing packets
-T|--tool <tool>                 The tool to use for the test
                                   Available Tools:
                                      iperf
                                      iperf3
                                      nuttcp
-t|--test_duration <seconds>     Duration for test (Default: 10)
-u|--udp                         Perform a UDP test
-w|--window <bytes>              TCP window size (Default: system default)
-W|--dynamic_window <bytes>      Dynamic TCP window fallback size (Default:
system default)
--tester_port <port>             For an endpoint-less test, use this port as
the server port (Default: tool specific)

Output Arguments
```

```
-d|--output_dir <directory>      Directory to save session files to (only if
-p)
-e|--facility <facility>         Syslog facility to log to
-f|--units <unit>                Type of measurement units to return
(Default: tool specific)
-p|--print                       Print results filenames to stdout (Default:
False)
-q|--quiet                       Silent mode (Default: False)
-r|--syslog_to_stderr            Send syslog to stderr (Default: False)
-v|--verbose                     Display verbose output
-x|--both                        Output both sender and receiver results
-y|--format <format>             Output format to use (Default: tool
specific)
--parsable                       Set the output format to the machine
parsable version for the select tool, if available

Misc Arguments
-h|--help                        Display the help message
-V|--version                     Show version number

Version: 1.5.5-1
```

Example usage of client

In the following example, we perform an iperf3 test between an endpoint and the client host.

```
[heiland@gw44 ~]$ /usr/bin/bwctl -T iperf3 -f m -t 10 -i 1 -c llnl-pt1.es.net
bwctl: Using tool: iperf3
bwctl: 50 seconds until test results available

SENDER START
Connecting to host 198.129.254.106, port 5027
[ 15] local 149.165.228.236 port 59438 connected to 198.129.254.106 port 5027
[ ID] Interval           Transfer     Bandwidth       Retr  Cwnd
[ 15]   0.00-1.00   sec  18.7 MBytes   157 Mbits/sec    0   3.42 MBytes
[ 15]   1.00-2.00   sec   148 MBytes  1237 Mbits/sec    1   11.2 MBytes
[ 15]   2.00-3.00   sec   181 MBytes  1520 Mbits/sec    3   9.80 MBytes
[ 15]   3.00-4.00   sec   192 MBytes  1615 Mbits/sec    0   11.3 MBytes
[ 15]   4.00-5.00   sec   188 MBytes  1573 Mbits/sec    0   11.1 MBytes
[ 15]   5.00-6.00   sec   195 MBytes  1636 Mbits/sec    0   11.4 MBytes
[ 15]   6.00-7.00   sec   179 MBytes  1499 Mbits/sec    0   10.1 MBytes
[ 15]   7.00-8.01   sec   188 MBytes  1559 Mbits/sec    2   10.6 MBytes
[ 15]   8.01-9.00   sec   199 MBytes  1681 Mbits/sec    0   11.3 MBytes
[ 15]   9.00-10.00  sec   196 MBytes  1647 Mbits/sec    0   8.26 MBytes
- - - - - - - - - - - - - - - - - - - - - - - - -
[ ID] Interval           Transfer     Bandwidth       Retr
[ 15]   0.00-10.00  sec  1.64 GBytes  1412 Mbits/sec    6             sender
[ 15]   0.00-10.00  sec  1.64 GBytes  1410 Mbits/sec
receiver

iperf Done.

SENDER END
```

Tools to help prepare for FPVA

In this section, we simply explore some basic shell commands an analyst might use to get an overview of the code – especially its multiple processes, in preparation for the FPVA.

```
.../perfsonar/github/bwctl$ ls -m
CHANGES, ChangeLog, DEVREADME, I2util/, INSTALL, LICENSE, Makefile,
Makefile.am, Makefile.in, README, README.md, RELEASE.TODO, TODO,
aclocal.m4, autom4te.cache/, bootstrap.sh*, bwctl/, bwctl.spec,
bwctl.spec.in, bwctld/, bwlib/, conf/, config/, config.log,
config.status*, configure*, configure.ac, contrib/, doc/, foo.ls,
libtool*, test/
```

The primary subdirectories containing the code base include:
`bwctl/, bwctld/, bwlib/`

In addition, the `I2util/` subdirectory is the Internet2 (I2) Utility library and contains:
Originally:
* error logging
* command-line parsing
* threading

perfSONAR added:
* random number support
* hash table support

To look for processes within the code, we can begin by using *grep* to look for `main()` programs or `fork()`'d processes:

```
..../perfsonar/github/bwctl$ grep "main(" bw*/*.c
bwctl/bwctl.c:main(
bwctld/bwctld.c:main(int argc, char *argv[])
bwlib/rijndael-test-fst.c:int main(void) {
```

```
.../perfsonar/github/bwctl$ grep 'fork()' bw*/*.c |grep '='
bwctl/bwctl.c:    pid = fork();
bwctld/bwctld.c:    pid = fork();
bwctld/bwctld.c:    pid = fork();
bwctld/bwctld.c:    pid = fork();
bwctld/bwctld.c:        mypid = fork();
bwlib/endpoint.c:    ep->child = fork();
bwlib/endpoint.c:    ep->child = fork();
bwlib/util.c:    pid = fork();
```

After manually inspecting the code a bit, we find all instances of where system commands will be executed:

```
.../perfsonar/github/bwctl/bwlib$ grep -i execcommand *.c
iperf.c:    n = ExecCommand(ctx, buf, sizeof(buf), cmd, "-v", NULL);
iperf3.c:    n = ExecCommand(ctx, buf, sizeof(buf), cmd, "-v", NULL);
nuttcp.c:    n = ExecCommand(ctx, buf, sizeof(buf), cmd, "-V", NULL);
owamp.c:    n = ExecCommand(ctx, buf, sizeof(buf), owping_cmd, "-h", NULL);
owamp.c:    n = ExecCommand(ctx, buf, sizeof(buf), owampd_cmd, "-h", NULL);
paris-traceroute.c:    n = ExecCommand(ctx, buf, sizeof(buf), traceroute_cmd,
"127.0.0.1", NULL);
ping.c:    n = ExecCommand(ctx, buf, sizeof(buf), ping_cmd, "-c", "1",
"127.0.0.1", NULL);
ping.c:    n = ExecCommand(ctx, buf, sizeof(buf), ping6_cmd, "-c", "1",
"::1", NULL);
tools.c:    n = ExecCommand(ctx, buf, sizeof(buf), cmd, "-h", NULL);
tracepath.c:    n = ExecCommand(ctx, buf, sizeof(buf), tracepath_cmd, NULL);
tracepath.c:    n = ExecCommand(ctx, buf, sizeof(buf), tracepath6_cmd, NULL);
traceroute.c:    n = ExecCommand(ctx, buf, sizeof(buf), traceroute_cmd,
"127.0.0.1", NULL);
traceroute.c:    n = ExecCommand(ctx, buf, sizeof(buf), traceroute6_cmd,
"::1", NULL);
util.c:ExecCommand(
util.c:        BWLError(ctx,BWLErrFATAL,errno,"ExecCommand():pipe(): %M");
util.c:        BWLError(ctx,BWLErrFATAL,errno,"ExecCommand():pipe(): %M");
util.c:        BWLError(ctx,BWLErrFATAL,errno,"ExecCommand():fork(): %M");
util.c:        snprintf(buf,sizeof(buf)-1,"ExecCommand(): exec(%s)",command);
util.c:                "ExecCommand(): waitpid(), rc = %d: %M",rc);
util.c:                    "ExecCommand(): %s exited due to signal=%d",
util.c:        BWLError(ctx,BWLErrWARNING,errno,"ExecCommand(): %s unusable",
command);
```

With this information, one would then analyze the `util.c:ExecCommand` function which does a `fork()`, redirects output to pipes, and does an `execvp`.