Interested in learning more about security?

# SANS Institute InfoSec Reading Room



## Protecting Insecure Programs

In most computer systems used today, programs are run that come from a variety of sources: the computer vendor, third-party vendors, open-source projects, consultants, and employees. In some cases, source code is available, but it is more often not. These programs often implement services that are made available to the general public, or listen on networks where all the participants are not trusted. Program flaws, such as buffer overflows, heap overflows, format bugs, and input validation bugs, enable attacks upon comp...

Shawn Instenes
GSEC option 1 version 1.4b
28 May 2003

**Protecting Insecure Programs**

*Introduction*

In most computer systems used today, programs are run that come from a variety of sources: the computer vendor, third-party vendors, open-source projects, consultants, and employees. In some cases, source code is available, but it is more often not. These programs often implement services that are made available to the general public, or listen on networks where all the participants are not trusted. Program flaws, such as buffer overflows, heap overflows, format bugs, and input validation bugs, enable attacks upon computing resources and a subversion of control. It is difficult to see how data integrity or availability can be maintained in such an environment. In order to mitigate the risks associated with the loss of control of computer programs, administrators of computer systems can apply defenses intended to limit damage and alert staff to trouble as early as possible. This document will examine several strategies to protect programs from malicious input, so that they will, in the worst case, abort processing rather than cause malicious code to be executed. Only host-based defenses are under consideration in this document. Particular attention will be spent on defenses appropriate to the Sun Solaris, Linux, and OpenBSD operating systems.

*Background*

When a new computer program is written, a decision must be made about which language it will be written in. Most programs running today have some sort of dependence on either the C or C++ computer languages- they are either written in one of these languages, or use libraries written in them. The choice of C or C++ is usually based two considerations: performance, and compatibility with existing code libraries. The C family of languages are structured in a way that is closely aligned with the design of most microprocessors, and they have been so widely ported to different architectures that it is difficult to find one in which they have not been ported.

However, the machine-oriented character of C is one reason why security vulnerability programming mistakes are so common [SecProg]. C reveals as much of the power of the underlying microprocessor as possible, but in so doing, also creates a support burden with the programmer. The standard C language has very few run-time checks that will trap and report common programming errors. The price for C's performance is the tedious diligence that must be taken in order to make the programs written in it reliable.

The pitfalls awaiting the C programmer are many. The largest single category of programming error that results in an exploitable vulnerability is the buffer

overflow. Year after year, roughly one-fifth [Icat] of all published vulnerabilities are buffer overflows. A buffer is a region of allocated memory that is used to store a series of identically sized data items. A buffer overflow, also known as a "stack smash" or "stack overrun", is an error in which the length of external data is not checked to make sure it will fit into a buffer that has been placed on the run-time stack. The C language does not enforce that the data written actually fit within the buffer space- the programmer must manually check this with code. As you might imagine, this check is usually omitted due to speed considerations or out of an ignorance of how damage could result from the flaw being exploited. This kind of vulnerability has been well understood for quite some time [Aleph1], yet additional examples of it are found weekly and then announced on mailing lists such as BugTraq [BugTraq]. Another kind of flaw, the heap overflow [w00w00], is similar to a buffer overflow but is less common and affects dynamically allocated data and pointers to data rather than the run-time stack; since heap overflows are less well known even experienced programmers neglect the checks that must be made to avoid creating this flaw in a program. Format bugs [Newsham] arise out of neglecting to carefully check external data given to the program, and treating it instead as printf(3) data; it is easy to forget about the %p and %n formats and what an attacker can do with them if they are allowed to supply the format string. On any given computer, there are very likely one or more programs that are vulnerable to the exploits written to take advantage of the vulnerabilities left behind by these simple programming errors.

Yet there are defenses that are effective. Even if the source code is not available, protections can be applied to make an attacker's job more difficult. These defenses can be divided into seven categories, with the first four applicable to all programs and the last three applicable to programs where the source code is available:

- Altering program memory maps
- Hardening system calls
- Filtering system calls
- Using a virtual machine
- Checking the source
- Compiler modifications
- Using libraries or languages with security guarantees

In the following discussion, each defense will be evaluated for effectiveness, ease of use, and portability. Not all defenses are appropriate for every program or situation.

*If you don't have source code*

If you do not have source to the program you're trying to protect, there are still things that can be done to limit the damage that can be done by an attacker. To exploit a vulnerable program, attackers make assumptions about the running environment of the program. By altering the running environment so that the attacker's assumptions are no longer valid, a deployed defense stops the attack

from succeeding.

*Altering program memory maps*

Each page in a computer system's memory has a set of permission bits describing what may be done with the page; on POSIX-oriented[i] systems these are named PROT_EXEC, PROT_READ, and PROT_WRITE; they indicate execute, read, and write permission, respectively. The memory management unit of the computer, in conjunction with the kernel, implements these protections. It is important to know that not all computer architectures implement the full set. For example, on x86 architectures, the hardware does not fully implement PROT_EXEC- PROT_READ implies PROT_EXEC.

The protection is implemented by modifying the default protection bits applied to a program's stack, and additionally other memory regions. For example, it is also appropriate to grant only PROT_READ permissions on a program's static data area, in case there is data within the program that an attacker could use as executable code or data that would affect the flow of execution if overwritten. Removing PROT_EXEC permission from the heap data area of a program makes heap-based exploits much more difficult.

This countermeasure requires no changes to the protected programs, and is a useful first line of defense. If it is successful in thwarting an exploit, the vulnerable program will cause a protection fault and terminate. It has no performance impact on the protected programs themselves, but may incur overhead within the operating system (this is system dependent). It requires a modification to the operating system, so this protection is not portable.

Solaris systems have a setting in /etc/system that removes execute permission on program stacks that has been available since version 2.6 [noexec]. It is enabled by adding a line in the file /etc/system that reads:

set noexec_user_stack = 1

Please note that this only protects the stack, not the heap or program data. 64-bit versions of Solaris have this protection on for 64-bit programs by default, however another related setting (noexec_user_stack_log) that enables logging of the attempts to use executable code on a program stack is not enabled by default. Both settings are recommended.

Linux systems can have their stack protected with the Openwall [Openwall] or grsecurity [grsecurity] patches. With the current Linux kernel the grsecurity patch is recommended, as it also includes a workaround on architectures that do not support the PROT_EXEC privilege correctly; this includes nearly all x86-based chips.

OpenBSD [OpenBSD] (version 3.3 and beyond) protects all of a program's memory regions appropriately, assuming that the architecture supports it,

completely automatically. A workaround for x86-based chips has been promised for version 3.4.

Most published exploits that rely on stack buffer overflows are stopped by this protection, but some exploits [Bypass] such as return-into-libc [Nergal] will still function.

*Hardening System Calls*

One type of protection for systems running untrusted or unreliable code is limiting the system calls that a program is able to invoke. Since all programs must use system calls to transfer data, open files, or modify file system objects, focusing on this interface allows untrusted programs to execute while limiting the damage they can do. The chroot(2)[ii] system call is an old example of this kind of protection; once a program is placed in a chroot "jail", it does not have the ability to reference any file system objects outside of the jail. Chroot, however, does not affect a program's ability to affect any resource not named by the file system, so it is limited in how well it can protect the system.

On Linux systems, chroot can be augmented with restrictions on capabilities, using the compartment [Heuse] tool. POSIX.1e capabilities [linuxcap] are an attempt to define work that a privileged process may do, and then limit each program so that it is capable of doing its assigned work but no other privileged system calls. The compartment program builds on the protections of chroot to further restrict what a program running in the compartment may do.

Kernel-loadable modules, on systems that support it, can be used to extend the protections surrounding untrusted programs. These additional protections further limit the damage that can be done by a subverted program. For example, on many systems the chroot call may be effectively canceled by a program with root privilege by using chroot twice more, once on a subdirectory of the chroot space, and lastly on the real system root. A kernel-loadable module can prevent this from occurring by simply disallowing a successful chroot for any process that already has a chroot in effect. Hardening the kernel has a minimal impact on most running programs, but is not portable. The protection is applied to all programs running, regardless if they are trusted or not. When this defense successfully stops an attack, the system calls invoked by the exploit fail. If the exploit corrupts the running environment, it is likely that the vulnerable program will terminate.

On Solaris-based systems, Papillon [Roqe] is a security-enhancing kernel module that protects against several well-known types of system exploits involving symbolic and hard file system links, limits what processes an unprivileged account may see, and further limits what a program under chroot(2) may do.

On Linux systems, there are several packages that implement enhanced-security features in the kernel. Among them is the Openwall patch [Openwall], the LIDS

project [LIDS], and the grsecurity [grsecurity] patch. In particular, the grsecurity patch implements both this protection and also protects program memory regions by making the stack and data areas not executable.

*Filtering System Calls*

A more powerful approach is to run untrusted programs under the watch of a monitoring process, and require that system calls invoked by the untrusted program be inspected and approved by the monitor program before being allowed to continue. This is more powerful in that the monitoring program may make decisions about the validity of system calls by knowing in advance what the untrusted program is supposed to do, where it is expected to manipulate files, whether it is expected to open or listen to network connections, and so on. Valid behavior of the untrusted program is coded in a profile of some kind, which is referenced when the untrusted program executes. An example of this form of defense is the Janus [Janus] project, which implements a simple version of a monitoring program which works on Solaris operating systems. Later work in the project enabled the monitor to work on Linux systems with the addition of a loadable kernel module. This defense will affect the performance of programs run under the watch of a monitor, but affect no other programs. The performance impact can be substantial (up to 40% in some cases) however, as every system call made by the untrusted program is examined by another user program. It is also a technology that is not widely portable, although commercial implementations of this idea exist for multiple platforms. Some commercial implementations are WireX Communications' SubDomain technology [WireX], which implements the monitor in the kernel, VXE [VXE], Entercept Communications' Entercept [Entercept], and Argus Systems' Pitbull LX [Argus].

*Using a virtual machine*

It is possible to emulate an entire operating system or computer at reasonable speeds. Therefore, it is now a practical defense to run the untrusted program from a virtual computer, as long as performance is not critical; this defense requires more compute resources than any other listed, even significantly more than filtering system calls. A virtual computer has allocated to it certain resources (memory, hard drive space, virtual network interface, et cetera) that it can access, but no more- it is completely unable to affect the computer that the virtual machine emulator is run from, barring errors in the implementation of the virtual computer. The entire set of system calls available to the untrusted program are emulated. On Linux systems, a project named User Mode Linux [uml] can present a virtual environment for a program to run in. It is important to note that while this is excellent protection for the real computer, the virtual one can still be compromised, with all of the data integrity issues that implies. However, since the entire running state of the virtual computer amounts to a single file on the physical computer, restoring the state of the virtual computer may amount to copying from a backup file and rebooting the virtual computer. One commercial product that implements a virtual machine is called VMWare [VMWare], which emulates the entire computer, including ROMs and boot

devices. Isolation between the real and virtual systems is so complete that entirely different "client" operating systems can be run within the virtual computer, and multiple simultaneous virtual computers can be run on a computer with sufficient storage and memory.

*If you have source code*

The following protections are possible only if source code to the vulnerable program is available. All of the previous protections are also applicable, of course.

*Checking the source*

If all source code were checked carefully for potential problems, then the need for the rest of these defenses would diminish greatly. However, it is important to list this as a defense, as programming flaws that are corrected before the program is used do not become vulnerabilities to be exploited later. There are many tools that programmers can use to help them do static source code analysis on their programs and find potential problems. One such program, Splint [splint] is a program that checks source code for potential security problems and annotates the source code accordingly, so that the problem in the code can be addressed before the program is used. A commercial product that supports multiple platforms is CleanScape Corporation's Lintplus [lintplus], which is a more general source code-checking tool.

*Compiler modifications*

Many exploits succeed because there is no way to detect a maliciously modified stack or data area without help from the compiler. A set of protections based in the compiler can make exploiting vulnerable programs in an undetectable way very difficult. A simple form of this protection, the stack canary (named so after canaries used in the mining industry, who fall faint or die at the slightest sign of noxious gas, giving warning to miners), is put on the stack by the subroutine entry code and verified by the subroutine exit code generated by the compiler. If the canary has been modified, then the exit code terminates the program with an error. Two schemes for generating stack canary values exist. The first uses a known value that will cause most string-handling functions to halt if encountered (the value is 0x000aff0d, developed by WireX). This is useful to protect against accidental problems, but fails in the face of an attacker who knows the canary value. Stack canaries are best generated by a source of true randomness, in order to make it difficult for an exploit writer to guess what the canary will be in advance; such a source of randomness may not be available.

A further protection that can be implemented in a compiler is the randomization of variables and code positions in memory, particularly the randomization of the location of loaded libraries. This defense may also require the help of the operating system; OpenBSD 3.3 does this by default. On Linux systems, the grsecurity [grsecurity] patch implements Address Space Layout Randomization

(ASLR), which randomizes the load position of the various memory allocations within an ELF binary. Both OpenBSD and the grsecurity patch further cause the addresses of the loaded libraries contain a NULL (zero) byte, so that attacks such as string format bugs are made even harder to exploit because these attacks cannot send a NULL byte through the vulnerable string functions.

Compilers are available that generate code that is resistant to buffer overflow attacks, and at least one compiler that will additionally rearrange the positions of variables, code, and libraries such that no address within an executing program will be known by an attacker. Examples of the former kind of compiler include StackGuard, StackShield [WireX], and the protection afforded by Microsoft .NET compilers if the /gs flag is used during compilation [Microsoft]. The SSP [SSP] compiler implements all of the mentioned protections.

Stack canaries alone can detect and (with appropriate function termination code) block most of the attacks based on buffer overflows. If the compiler further reorders stack variables and load locations of libraries, an exploit will be unable to construct a valid stack frame (with a high degree of probability, based on the size and randomness of the stack canary).

*Using libraries or languages with security guarantees*

There are libraries of code that aid in avoiding problems with buffers, pointers, and memory management; using them requires some porting effort but in high-risk applications the time would be well spent. One of these libraries, SafeStr [SafeStr], provides a consistent and safe interface to string-handling functions, the root of many security vulnerabilities. Of course, effort is required to re-code any string handling that the program may do, converting it to use the new library.

Another code library, libsafe [libsafe], can protect programs that have already been compiled if that program uses dynamically linked libraries. No source code changes need to be made, however it can only protect against exploits directed at vulnerable functions in the C library, not all vulnerable functions. This protection is fairly portable, as most operating systems today have some concept of a shared object library that is linked to programs at run time.

If the program is put into a particularly high-risk situation (a machine without a firewall, processing sensitive data), it may be prudent to consider porting or implementing the program in a language that has some security guarantees. For example, Java programs are guaranteed to never be susceptible to buffer overflows, barring an error in the implementation of the Java run-time library. This is by design, as string handling objects of two types are implemented in the language, one for static strings and another for string buffers, which perform bounds checking on every modification. Pointers to arbitrary memory locations simply do not exist in Java as they do in C, which eliminates another class of errors and vulnerabilities. Other languages, such as Scheme and CAML, also have security guarantees. The security “stance” of a computer language may become an important consideration with future Windows programming, as

Microsoft appears to be developing a CAML-based language for .NET development [Fsharp].

Time and money constraints are often a barrier to creating secure programs, and adding security after the program is already written generally consumes too many resources to be a serious consideration, especially when someone suggests porting a program to a completely different language. There are dialects of C that promote enhanced security that may be useful to pursue in this case. Ccured [ccured] is a source code translator that, given an input C language program with appropriate hint markup, creates a new version of the program that contains checks that prevent memory safety violations. The resulting program runs more slowly, but will always catch attempts to exploit stack and heap overflows. Another approach, a dialect of C called Cyclone [cyclone], approaches the problem of unsafe memory accesses by implementing built-in safe pointer types, exception handling, and a robust region-based memory-management scheme that eliminates problems with dynamically allocated structures. The disadvantage is that a good portion of the C code will have to be re-written in correct Cyclone to take advantage of the memory model and exceptions in order to be protected.

*Conclusion*

There are all kinds of production computing environments. It is a rare facility that is able to run a program it truly trusts, and even rarer for a facility that does not have to run any untrusted programs. Most businesses have a need to present an image to the Internet, which is of course the world's largest untrusted computer network. It is reassuring to know that there are ways to protect untrusted programs, even those programs without source code, such that any resulting damage from active attack will be extremely limited. The available protections are such that the worst that an attacker may do is cause a denial of service, which cannot be completely protected against in any event. If several of the methods detailed in this document could be used simultaneously, the resulting "defense in depth" environment would be very difficult to exploit.

There are several avenues for further work that may be fruitful. For example, the defense "filtering system calls" is possibly the most powerful if source code is not available, since it operates from an assumption that allowed behavior can be codified and all other program behavior should be denied. However, it is also one of the more resource-intensive defenses. Improving the performance of this defense would quite definitely make its use more widespread. The virtual machine defense is also quite promising, if automated checking, backup & restoration of virtual machine state can be implemented.

Solaris systems can be defended with the noexec_user_stack setting, without any other modifications but will still be vulnerable to large classes of attacks. It would be helpful if Sun would implement the ASLR support available in OpenBSD and Linux with grsecurity, so that attacks not executing within the stack would be difficult to execute. The Papillon [Roqe] module can be used so

that some attacks can be stopped without impacting performance. Lastly, Solaris systems can also be protected by the defenses in the SSP [SSP] compiler if it can be used to compile programs from source.

Linux users can apply the grsecurity patch and gain many of the defenses mentioned in this document- stack and data areas are made non-executable, programs are mapped into memory using ASLR, and system calls are hardened against potential exploits. With the addition of a compiler that generates code with stack protection, Linux computers can be well defended against attack.

Users of OpenBSD 3.3 appear to have the best set of defenses available, and further do not have to patch or modify the system to enjoy the benefits. OpenBSD uses several defenses (altering memory maps, modified compilers, hardened system calls, safe function libraries) in order to make an attackers task as difficult as possible. While no other operating system does as well, at least a few of these defenses can used to help secure any program that needs to be used in the enterprise.

*References:*

[Cyclone] Jim Trevor; Grossman, Dan; Hicks, Michael; Cheney, James; Wang, Yanling. "Cyclone: A safe dialect of C." 21 Nov 2001. URL: http://www.research.att.com/projects/cyclone/papers/cyclone-safety.pdf

[Openwall] Openwall Project. 19 May 2003. URL: http://www.openwall.com/linux/

[SSP] IBM. SSP/ProPolice. 26 May 2003. URL: http://www.trl.ibm.com/projects/security/ssp/

[noexec] Sun Microsystems. "Solaris Tunable Parameters Reference Manual." 26 May 2003. URL: http://docs.sun.com/db/doc/806-7009

[Roqe] Rieck Konrad. Papillon. 26 May 2003. URL: http://www.roqe.org/papillon/

[Bypass] WindowSecurity.com. "How to Bypass Solaris Nonexecutable Stack Protection". 26 May 2003. URL: http://secinf.net/unix_security/How_to_bypass_Solaris_nonexecutable_stack_protection_.html

[Microsoft] Microsoft. "/gs flag". 26 May 2003. URL: http://go.microsoft.com/fwlink/?LinkId=7260

[SafeC] Cheng Wanli, Gandhi Denis, Wadekar Vihar. "Safe C". 26 May 2003. URL: http://www.doc.ic.ac.uk/~dg100e/home.htm

[Vuln02] Arce Ivn. "Multiple Vulnerabilities in stack smashing protection

technologies." 26 May 2003. URL:
http://www.linuxsecurity.com/articles/server_security_article-4869.html

[BOM] Gerhart, Dr. Susan. Buffer Overflow Demos, 26 May 2003. URL:
http://nsfsecurity.pr.erau.edu/bom/

[VXE] InteS. VXE. 26 May 2003. URL: http://www.intes.odessa.ua/vxe/

[Janus] Wagner David, Garfinkel Tal. Janus Project. 26 May 2003. URL:
http://www.cs.berkeley.edu/~daw/janus/

[WireX] WireX. StackGuard and StackShield. 26 May 2003. URL:
http://www.immunix.org/stackguard.html

[Aleph1] Aleph One. "Smashing the stack for fun and profit." 8 Nov 1996. URL:
http://www.phrack.org/phrack/49/P49-14

[LIDS] Xie Huagang, Biondi Philippe. LIDS Project. 26 May 2003. URL:
http://www.lids.org/

[SELinux] National Security Agency. Security Enhanced Linux. 26 May 2003.
URL: http://www.nsa.gov/selinux/

[uml] Dike Jeff. User Mode Linux. 26 May 2003. URL: http://www.user-mode-linux.sourceforge.net/

[Icat] National Institue for Standards and Technolgy. 26 May 2003. URL:
http://icat.nist.gov/icat.cfm?function=statistics

[BugTraq] BugTraq mailing list. 26 May 2003 URL:
http://www.securityfocus.com/archives/1

[Entercept] Entercept Communications. "Entercept." 26 May 2003. URL:
http://www.entercept.com/

[Vmware] Vmware Corporation. Vmware. 26 May 2003. URL:
http://www.vmware.com/

[Argus] Argus Systems. Pitbull LX. 26 May 2003. URL: http://www.argus-systems.com/

[SafeStr] Messier Matt, Viega John. "Safe C String Library 0.9.5" 19 May 2003.
URL: http://www.zork.org/safestr/

[w00w00] Conover Matt, w00w00 Security Development. "w00w00 on Heap Overflows." Jan 1999. URL: http://www.w00w00.org/files/articles/heaptut.txt

[OpenBSD] de Raadt Theo, many others. OpenBSD version 3.3. 1 May 2003.

URL: http://www.openbsd.org/33.html

[libsafe] Avaya Lab Research. Libsafe. 26 May 2003. URL: http://www.research.avayalabs.com/project/libsafe/

[Fsharp] Peterson Robyn. "Inside Microsoft's New F# Language." 22 May 2003. URL: http://www.extremetech.com/print_article/0,3998,a=42282,00.asp

[ccured] Necula George, McPeak Scott, Weimer Westley, Harren Matthew, Condit Jeremy. "Ccured Documentation." 26 May 2003. URL: http://manju.cs.berkeley.edu/ccured/

[SecProg] Wheeler David A. "Secure Programming for Linux and Unix HOWTO." 3 March 2003. URL: http://www.dwheeler.com/secure-programs/Secure-Programs-HOWTO.html

[Heuse] Heuse Marc. Compartment. 24 Feb 2001. URL: http://www.suse.de/~marc/compartment.html

[linuxcap] Morgan Andrew G. "Linux-Privs." 21 Apr 1997. URL: http://www.linuxsecurity.com/resource_files/server_security/linux-privs/linux-privs.html

[Nergal] Nergal. "The advanced return-into-lib(c) exploits." URL: http://www.phrack.org/show.php?p=58&a=4

[Newsham] Newsham Tim. "Format String Attacks." 11 Sept 2000. URL: http://julianor.tripod.com/tn-usfs.txt

[grsecurity] Grsecurity patch. 9 May 2003. URL: http://www.grsecurity.net/

[splint] Evans David, Larochelle David, Baker Chris, Friedman David, Lanouette Mike, Phan Hien. Security Programming Lint. 26 May 2003. URL: http://lclint.cs.virginia.edu/

[lintplus] CleanScape Corporation. Lintplus. 26 May 2003. URL: http://www.cleanscape.net/products/lintplus/index.html

[i] POSIX is a set of standards published by the IEEE at http://www.pasc.org/. It stands for "Portable Operating System Interface".

[ii] On many POSIX-compatible systems, it is customary for "man" (manual) pages to be separated into sections, identified by number. Section 1 covers user commands, and section 2 details system calls. The number in the parentheses indicates which section is relevant to the topic being discussed. This disambiguates chroot(1), which is a command that may be typed into a shell prompt, from chroot(2) which details how this system call is invoked and what it does.

Last Updated: October 23rd, 2017

# Upcoming SANS Training

**Click Here for a full list of all Upcoming SANS Events by Location**

| | | | |
|---|---|---|---|
| SANS Seattle 2017 | Seattle, WAUS | Oct 30, 2017 - Nov 04, 2017 | Live Event |
| SANS Gulf Region 2017 | Dubai, AE | Nov 04, 2017 - Nov 16, 2017 | Live Event |
| SANS Amsterdam 2017 | Amsterdam, NL | Nov 06, 2017 - Nov 11, 2017 | Live Event |
| SANS Milan November 2017 | Milan, IT | Nov 06, 2017 - Nov 11, 2017 | Live Event |
| SANS Miami 2017 | Miami, FLUS | Nov 06, 2017 - Nov 11, 2017 | Live Event |
| SANS Paris November 2017 | Paris, FR | Nov 13, 2017 - Nov 18, 2017 | Live Event |
| Pen Test Hackfest Summit & Training 2017 | Bethesda, MDUS | Nov 13, 2017 - Nov 20, 2017 | Live Event |
| SANS Sydney 2017 | Sydney, AU | Nov 13, 2017 - Nov 25, 2017 | Live Event |
| GridEx IV 2017 | Online, | Nov 15, 2017 - Nov 16, 2017 | Live Event |
| SANS San Francisco Winter 2017 | San Francisco, CAUS | Nov 27, 2017 - Dec 02, 2017 | Live Event |
| SANS London November 2017 | London, GB | Nov 27, 2017 - Dec 02, 2017 | Live Event |
| SIEM & Tactical Analytics Summit & Training | Scottsdale, AZUS | Nov 28, 2017 - Dec 05, 2017 | Live Event |
| SANS Khobar 2017 | Khobar, SA | Dec 02, 2017 - Dec 07, 2017 | Live Event |
| SANS Austin Winter 2017 | Austin, TXUS | Dec 04, 2017 - Dec 09, 2017 | Live Event |
| SANS Munich December 2017 | Munich, DE | Dec 04, 2017 - Dec 09, 2017 | Live Event |
| European Security Awareness Summit & Training 2017 | London, GB | Dec 04, 2017 - Dec 07, 2017 | Live Event |
| SANS Bangalore 2017 | Bangalore, IN | Dec 11, 2017 - Dec 16, 2017 | Live Event |
| SANS Frankfurt 2017 | Frankfurt, DE | Dec 11, 2017 - Dec 16, 2017 | Live Event |
| SANS Cyber Defense Initiative 2017 | Washington, DCUS | Dec 12, 2017 - Dec 19, 2017 | Live Event |
| SANS Security East 2018 | New Orleans, LAUS | Jan 08, 2018 - Jan 13, 2018 | Live Event |
| SANS SEC460: Enterprise Threat Beta | San Diego, CAUS | Jan 08, 2018 - Jan 13, 2018 | Live Event |
| SANS Amsterdam January 2018 | Amsterdam, NL | Jan 15, 2018 - Jan 20, 2018 | Live Event |
| Northern VA Winter - Reston 2018 | Reston, VAUS | Jan 15, 2018 - Jan 20, 2018 | Live Event |
| SEC599: Defeat Advanced Adversaries | San Francisco, CAUS | Jan 15, 2018 - Jan 20, 2018 | Live Event |
| SANS San Diego 2017 | OnlineCAUS | Oct 30, 2017 - Nov 04, 2017 | Live Event |
| SANS OnDemand | Books & MP3s OnlyUS | Anytime | Self Paced |