Running Malicious Code By Exploiting Buffer Overflows: A Survey Of Publicly Available Exploits

Ari Takanen, Marko Laakso, Juhani Eronen, Juha Röning
University of Oulu, Department of Electrical Engineering, Finland

About the Authors

Ari Takanen is an undergraduate student at the University of Oulu. He comes from an NT networking and administration background, but is now concentrating on vulnerability research. His goal is a master's degree in Information Engineering (MSEE). Meanwhile he is working as a research scientist at the Department of Electrical Engineering.

Marko Laakso has a UNIX background gained from administrating the network of the Department of Electrical Engineering at the University of Oulu. Since 1996 he has been pestering AusCERT and software vendors with security bug reports that are too long to be read.

Juhani Eronen is studying Information Engineering at the University of Oulu, aiming at a master's degree - and beyond. His current research in the group is related to secure programming and test design, and his studies are focused on computer theory, mathematics and other abstractions.

All three are carrying out research in the Secure Programming Group (OUSPG) led by Professor Juha Röning, who has been a member of the Department of Electrical Engineering at the University of Oulu since 1983, having received his master's degree (MSEE) from there in 1983, a Licentiate in Technology with honors in 1985, and a Doctorate in Technology in 1992. He is a member of SPIE, IEEE, Sigma Xi, the Finnish Pattern Recognition Society, and the Finnish Artificial Intelligence Society (FAIS).

Mailing Address: University of Oulu, Department of Electrical Engineering, Computer Engineering Laboratory, PL 4500, FIN-90401 Oulu, Finland; Phone: +358-8-553-2812; Fax: +358-8-553-2612; E-mail: ouspg@ee.oulu.fi.

Descriptors

information security, software quality, secure programming, implementation error, implementation level vulnerability, buffer overflow, exploit, malware, public disclosure, survey

Reference to this paper should be made as follows: Takanen, A., Laakso, M., Eronen, J. and Röning, J. (2000) 'Running Malicious Code By Exploiting Buffer Overflows: A Survey Of Publicly Available Exploits', EICAR 2000 Best Paper Proceedings, pp.158-180.

Edited by Urs E. Gatikker, © EICAR c/o TIM World ApS, Aalborg, Denmark, ISBN: 87-987271-1-7
www.papers.weburb.dk

Running Malicious Code by Exploiting Buffer Overflows: A Survey of Publicly Available Exploits

Abstract

Buffer overflow vulnerabilities emerge and are announced frequently. Exploitation details and exploits are publicly available for interested parties. These code-based attacks allow malicious code to be executed on the vulnerable systems. This paper provides a brief introduction to buffer overflow vulnerabilities and methods of exploiting them and a review of well-known articles on buffer overflows. A survey is made of exploits that are easily available to everyone, including the underground community. A database of exploits is chosen and studied. Articles on buffer overflows are reviewed, and the exploits presented are examined in terms of functionality. An attempt is made to analyze the distinctive characteristics and operational components of the exploits. Exploitation methods were observed to follow the same guidelines for all platforms. Some figures on exploits are shown. Protection methods and advanced exploitation methods for circumventing them are considered, and issues requiring future research or pertaining to the development of exploits are raised.

Introduction

Attack by Running Malicious Code as Low Level Active Content

A buffer overflow exploit is an attack in which the input string contains arbitrary and possibly malicious code that is executed once the boundary of a buffer has been exceeded. Although several publicly available exploits exist and guides on how they can be created are available, advisories from software vendors can sometimes include comments that exploitation of the overflow can be difficult and underestimate the risks involved in buffer overflow vulnerabilities (Microsoft, 1999a, b).

People are looking for vulnerabilities, and sometimes publicly announce their findings. Traditionally, only Unix operating systems have been under the scrutiny from the hacking community, but nowadays a wide variety of platforms have been hit by these exploits, revealing the real impact that insecure programming can have on widely distributed software. The exploits vary greatly in content. A popular payload for the exploit is a code that executes commands with the privileges of the target process. There are also some exploits such as the new eEye exploit against the Microsoft web-server, that download and execute a prepared file on the machine of the victim (eEye, 1999b). As new buffer overflow vulnerabilities are found and announced, new exploit variants emerge that exploit them.

An Implementation Level Vulnerability

A typical cause of information security vulnerabilities is a flaw introduced at some phase of the software development. Buffer overflows are one type of the 'implementation level vulnerabilities', which is to say that they are caused by mistakes made when implementing, i.e. coding, the software. By following the frequency of different vulnerabilities in public, it can be seen that these implementation level cases are the cause of a large portion of those that are disclosed. We have observed that implementation level vulnerabilities are often caused by insecure file manipulation functions (e.g. mktemp(), popen()), string manipulation functions without bounds checking (e.g. strcpy(), gets()) and careless execution functions (e.g. system(), execve()). We have noted from public disclosures that a large portion of the vulnerabilities that are found appear to be buffer overflows, usually caused by string manipulation functions.

Buffer Overflows in a Nutshell

A buffer overflow takes place when the memory reserved for a variable is exceeded and the application or operating system writes data outside this memory. Two categories can be recognized: stack and heap based overflows. The overflowing of the memory itself is similar in both, but they vary somewhat in the way they are exploited, due to different information located next to the memory. Function arguments, saved values of registers and local variables are stored in the stack of the thread. Variables with a security-critical content may also be stored in static memory, or in dynamic memory known as the heap.

As the stack contains not only the variables of vulnerable functions but also the return addresses for the points from which the functions were called up, overwriting enough data potentially gives the exploit full control over the process. As in most platforms the stack grows downwards in memory whereas the addressing scheme is incremental, the previously stored data is overwritten. If the stack were to grow upwards, the data following the stored variable would be overwritten thus causing the return address of the vulnerable library function itself to be overwritten, and is not a solution for the problem (Sommerfeld, 1997). Heap overflows are often misunderstood as being more difficult to exploit, as it requires some additional work, but in practice both are usually exploitable. And if the vulnerability is exploitable, it is always easily exploitable, as the exploitation details are often published, as will be shown later.

No Source Code for the Software

Several popular software packages are black-box applications distributed without source code, and thus their security can be assumed to be difficult to evaluate. For the users and maintainers this could create a need to avoid revealing the software packages that are in use, thus reducing the risks caused by the holes that maintainers might be unaware of. This might also cause a false feeling of security as, most networked software, for example, can be identified by the peer that it is communicating with. This information can also pass through a long chain of proxies, routers and other servers on its way through the net.

The security community and the underground community seem to have taken the approach that if programmers and software companies are unable to find the security holes in their software, then they will do it. As there are methods of evaluating the security of black-box software (Laakso, Takanen & Röning, 1999a), the administrators of security-critical systems can look for them before they are found by more mischievous people. Sub-cultures have appeared among young computer people which are dedicated to looking for holes and possibly also exploiting them, whereas some universities, research centers and software vendors may even see themselves to be inferior in practical skills to the underground community.

A Study of Buffer Overflow Exploits

This paper will present a detailed study of some known exploits that are available from the security community and consider their level of complexity and the possible threat they may pose. We will take a look at articles published on buffer overflow vulnerabilities and the exploitation details they present, and briefly mention the available testing methods and possible solutions. Some publicly available non-academic work will also be studied, e.g. exploitation guides with example exploits.

The case study presented here will concentrate on exploits available to the underground community and compare them with those known from the literature. An overall study of the sample exploits will be presented, and after a more thorough autopsy, their basic structure will be deduced. Terms for the sub-components will be collected or proposed for use in the analysis, and conclusions will be drawn from the case study. Also the development of

methods for protecting against buffer overflow vulnerabilities that are in progress and their possible effects in the next millennium will be reviewed and consideration will be given to possible improvements in the exploits that the future may bring.

Literature Review

Historical Research into Overflows

The day when the world finally acknowledged the risk entailed in overflow vulnerabilities and started coordinating a response to them was the day when the Internet Worm was introduced, spread and brought the Internet to its knees. In his analysis of the worm, Spafford (1989) presents some details of the buffer overflow exploit against the 'fingered' program for BSD-derived versions of Unix running on VAX computers. Some research and response work was initiated in the security field, but a limited amount of academic work was available to the general public regarding the causes behind buffer overflow vulnerabilities and their ease of exploitation. Since then, several discoveries have been announced as public disclosures on public disclosure mailing lists, such as Bugtraq, where security-aware people has donated their knowledge on the subject (Myers, 1993-1999; Netspace, 1995-1999; SecurityFocus, 1999).

Current Status of Research

Some research results have been published related to implementation level vulnerabilities. Garfinkel and Spafford (1996) present some guidelines on programming secure applications for Unix and networks, and research has been going on in other fields, such as the classification of vulnerabilities (Krsul, 1998a; Bishop, 1996) and the tracking of different kinds of vulnerabilities in databases (UC Davis Vulnerability Project; Herold, 1998; Krsul, 1998b). Bishop has contributed publications and tutorials on the teaching of secure programming (Bishop 1993, 1997), and a major contribution to exploitation details and prevention comes nowadays from the security community, which follows and tracks vulnerabilities, both privately within the vendors and publicly in mailing lists and Usenet newsgroups, for example. Our research group has published some past experiences of the process of reporting vulnerabilities (Laakso, Takanen & Röning, 1999b).

Buffer overflow vulnerabilities have traditionally been exposed by code-auditing methods, and research has been done into automation of the search for use of vulnerable functions and suspect blocks of code (L0pht, 1999). Some research has been carried out into testing for buffer overflow vulnerabilities by various black-box methods of automatic penetration (eEye, 1999a; Ghosh, Schmid & Shah, 1998; Miller et al., 1995) and white-box methods such as fault injection (Voas & McGraw, 1998). Overflows, and other implementation level vulnerabilities have been discovered through symbol interposition (Laakso, Takanen & Röning, 1999a). Some research into operating system and compiler development is also available, investigating new compiler improvements and kernel-level methods of protecting against them (Cowan, Pu & Hinton, 1998).

Excerpts From the Hacker's Bookshelf

A recent upheaval, which occurred not through demonstration in real-life but by education, came from an article by Aleph One (1996) in an underground publication that describes in detail the structure of the stack and how a stack-based buffer overflow can be exploited in the Linux operating system. Aleph One also presents a way to generate an exploit egg which contains what he calls a shellcode to spawn a shell when run. He provides guidelines for generating the shellcode by compiling a simple C-program using known system calls and extracting the assembly code from it with a debugger, and then modifying the assembly code. The egg is the wrapping around the shellcode that makes sure that it is executed. His paper presents working shellcodes for the x86/Linux, SPARC/Solaris and SPARC/SunOS platforms, and a generic eggshell program that creates the working exploit egg with different variable parameters such as offset, the size of the exploitable buffer and the size of the egg.

Several other articles have appeared since then providing guides on how to exploit buffer overflows, and dozens of exploit eggs, i.e. prepared exploitation codes, have been made available by the underground community. Articles that describe how to create an exploit are distributed on the Internet and are widely read, and the instructions are applied to generate new exploits.

Mudge (1995), from 'L0pht Heavy Industries', has also made his personal notes available on the Internet. These demonstrate a simple exploit egg directed against a libc/syslog overflow on BSDI, with enough details for using the guidelines against other programs and on other platforms.

Smith (1997) combines the previous articles, providing a thorough guide for exploiting buffer overflows in different Unix variants. Adopting a more academic approach, he manages to wrap up a good coverage of the history and terminology of this field, and mentions the publicity created by the advisories on buffer overflows that some organizations and groups have released. He gives a short explanation of Unix operating system properties such as SUID programs and Linux stack structure and functionality. Smith has collected together shellcodes for different processor architectures, including those published by Aleph One and shellcodes for AIX and HPUX, with no reference to their origin. Also presented is a discussion on secure programming, with a list of some of the vulnerable functions and more secure replacements for them. He also gives a short summary and review of various overflow solutions.

From the 'Cult of the Dead Cow', Dildog (1998) announced in the Bugtraq mailing list a thorough walk-through for exploiting Windows overflows, taking the Microsoft Netmeeting as the object of a case study. The value that this study adds to exploiting buffer overflows is that it presents a way of using the stack pointer, which usually already points at the exploit string itself, to initiate the exploit. The return address is pointed at any fixed address, whether in the exploitable program itself or some dynamically loaded library, that contains the assembly instructions needed to jump to the address that the stack pointer is pointing to. Therefore the exploit script that Dildog presents avoids the problem that arises

from the fact that the stack address can vary between threads or processes. Also, by using standard application programming interface functions, the exploit is able to download and run any file from the Internet.

Litchfield (1999) builds a simple shellcode for the Windows NT platform. He discusses stack-based buffer overflows in detail, and after a brief discussion of the structure of the process memory and stack in Windows NT, he takes rasman.exe as a case study and presents the symbolic assembly code that creates a local shell with elevated privileges.

In his tutorial on heap-based buffer overflows, Conover (1999), from the ‘w00w00 Security Development’ group, starts by saying: "Heap/BSS-based overflows are fairly common in applications today; yet, they are rarely reported." He notes that current protection methods, such as current non-executable stacks, do not guard against heap-based overflows, and presents detailed information on heap/BSS memory areas and their usage with vulnerable example programs, instructions on how heap overflows can be exploited, and case studies. He uses the shellcode from Aleph One and the 'exact offset' approach in the exploit egg, with consideration for traditional improvements used in popular stack-based exploits. He lists a dozen popular functions and programming methods that cause vulnerability to appear. His tutorial counters most claims regarding the insignificance of heap-based overflows by demonstrating their ease of exploitation.

Methods

The main effort in this work is put into studying the exploit eggs that are available and in use in the underground community. An attempt will be made to accomplish the following tasks:

- An archive of buffer overflow exploits will be chosen according to their availability and coverage. This archive should also be well organized for easier study. Only one should be selected, to reduce manual labor, due to lack of standardization in database formats.

- The exploit archive will be studied and some figures drawn to represent the number of entries on buffer overflows and exploits against buffer overflow vulnerabilities.

- The exploits will be studied to gather some knowledge of the origin of the freely available exploits and the possible purposes of publishing them.

- An attempt will be made to form distinctive groups of exploits with similar exploitation details.

- The structures of the most advanced exploit eggs in each group of exploits will be studied in terms of their complexity and future use. Details of the exploit eggs will be analyzed and summarized.

- The results will be studied in order to assess the general trend in the maturity of exploits.

Results

Exploit Databases

There are several sites that contain hacking guides and exploits. Some of them are public and maintained by various groups or private persons (Crvelin, Fyodor, Packet Storm, Rewted, Rootshell, SecurityFocus) and are organized in various ways, by operating systems, vendor or date, or just as a mixed listing. The Rootshell archive, www.rootshell.com, was chosen as a starting point for gathering exploits for further study. It has been maintained for a long time, and is well organized in terms of publication time, from 1997 onwards, whereas most other databases are organized by the target operating system. This provided us with a clear timeline for the exploits. It was also selected because it was publicly available and well known. Given the active collecting methods, the selection of exploits in the database probably represents sources other than just public disclosure lists such as Bugtraq.

The Rootshell exploit archive was studied and exploits directed against buffer overflow vulnerabilities were gathered from it. One drawback was that the database entries often lack the original media, such as the mailing list or newsgroup that the exploits were taken from, or even the authors of the exploits.

Exploit Entries in the Database

Only database entries that contained a fully operational exploit egg with source code were studied, other exploits being rejected. Similarly exploits that were obscured by packaging, e.g. in compressed form, were left outside the scope of this study. All in all, 95 exploits were studied, all of which are available at www.rootshell.com. The tables provided here present the following information on the exploits:

- Number: All exploits are numbered for indexing purposes.

- Date: The date refers to the addition of the exploit to the archive, and is not necessarily the date when the exploit first appeared. This was taken from the original archive listing.

- OS: An abbreviation for the operating system that the exploit is targeted at. This may be different from the attacking operating system. The following abbreviations are used:
  - A AIX
  - B BSDi, and possibly other BSD variants
  - D Digital Unix
  - FB FreeBSD
  - H HP-UX

- I IRIX
- L Linux
- OB OpenBSD
- S Solaris/SunOS
- SCO SCO Unix
- W WinNT, Win95, Win98

- Architecture: The processor architecture that the shellcode was designed against: (x86), (S)PARC, (RS)6000, (MIPS), (A)lpha

- Shellcode: The payload containing the arbitrary code for execution. This can contain any operations that the target program has access to, but as the name indicates, it usually provides a shell with the privileges of the target software. The following abbreviations are presented for the operation of the shellcode: (LS) local shell, (RS) remote shell, (LC) local commands and (RC) remote commands. (RX) is a special case in which the shellcode starts up a remote X-terminal. (O) denotes others described in detail later in this paper, and (?) denotes unknown shellcodes.

- Type: (LZ) is mentioned if the exploit egg depends on the traditional Landing Zone method presented by Aleph One, for example. Landing zone is an area of instructions that are irrelevant to the operation but make it possible to jump into a wider area that is not so dependent on the starting address of the shellcode in the memory.

- Restricted: This means that the permitted instruction set used in the shellcode is restricted by filtering performed on the string, for example.

- Non-exec: Entries marked as non-exec work even when non-executable stack protection is in use.

- Content: (ASM) is indicated if either the symbolic assembly instructions are provided in comments or the shellcode is compiled from the assembly instructions at the time of compilation of the exploit egg. Otherwise the shellcode is usually presented as a precompiled packet or string.

- Special: There is something unusual in the entry. The entry is marked with a letter, and the special feature of the exploit is explained below, and later in the paper. Each special case is explained only once, even though several similar entries existed.

- Description: The name of the vulnerable program.

1996: Of the 103 entries for 1996, 5 concerned buffer overflows, and 4 of these contained an exploit:

Table 1: Rootshell buffer overflow exploits from 1996

Number	Date	OS	Architecture	Shellcode	Type	Restricted	Non-exec	Content	Special	Description
01	08/26/96	L	x86	LS	LZ					'dip'
02	08/26/96	L, FB	x86	LS	LZ					'umount'
03	08/26/96	L	x86	LS	LZ					'xterm'
04	08/26/96	FB	x86	LS	LZ			ASM		'rdist'

1997: Of the 190 entries for the first six months of 1997, 34 concerned buffer overflows, and 31 contained an exploit:

Table 2: Rootshell buffer overflow exploits from the first six months of 1997

Number	Date	OS	Architecture	Shellcode	Type	Restricted	Non-exec	Content	Special	Description
05	01/01/97	S	S	LS	LZ					'rlogin'
06	01/30/97	S	S	LS	LZ					'rlogin'
07	01/30/97	B	x86	LS	LZ					'cron'
08	01/30/97	FB	x86	LS	LZ					'modstat'
09	02/11/97	S	S	LS	LZ					'ffbconfig'
10	02/11/97	S	S	LS	LZ			ASM		'minicom'
11	02/25/97	S	S	LS	LZ					'passwd'
12	02/28/97	FB	x86	LS	LZ					'pppd'
13	03/07/97	L	x86	LS	LZ					'SuperProbe'
14	03/13/97	FB	x86	O	LZ			ASM	A	'sendmail'
15	03/16/97	B, L	x86	RC	LZ	R		ASM	B	'in.talkd'
16	03/16/97	L	x86	LS	LZ	R		ASM		'crontab'
17	04/03/97	A	RS	LS	LZ			ASM		'mount'
18	04/28/97	L	x86	LS	LZ			ASM		'xlock'
19	05/15/97	L	x86	LS	LZ			ASM		'cxterm'
20	05/17/97	A	RS	LS	LZ			ASM		'dtterm'
21	05/26/97	A	RS	LS	LZ			ASM		'lquerylv'
22	05/26/97	I	MIPS	LS	LZ			ASM		'permissions'
23	05/26/97	I	MIPS	LS	LZ			ASM		'login'
24	05/26/97	I	MIPS	LS	LZ			ASM		'xlock'
25	05/26/97	I	MIPS	LS	LZ			ASM	C	'df'
26	05/29/97	I	MIPS	LS	LZ			ASM		'xterm'
27	05/29/97	I	MIPS	LS	LZ			ASM		'iwsh'
28	05/29/97	I	MIPS	LS	LZ			ASM		'printers'
29	06/10/97	A	RS	LS	LZ			ASM		'dtaction'
30	06/11/97	S	S				X		D	'ps'
31	06/15/97	I	MIPS	LS	LZ			ASM		'eject', 'xlock', 'pset', 'df', 'ordlist', 'xconsole'
32	06/20/97	L	x86	LS	LZ					'xgv'
33	06/20/97	L	x86	LS	LZ					'bdash'
34	06/24/97	L	x86	O	LZ				E	'imapd'
35	06/27/97	L	x86	LS	LZ					'smbmount'

The special entries here are:

A: 14: Executes several preset commands to create a setuid shell in /tmp
B: 15: A complex exploitation method requiring a compromised nameserver and additional tools
C: 25: A guide for exploiting IRIX operating system on the RISC processor architecture
D: 30: A complex heap overflow exploit
E: 34: The shellcode remotely modifies the password file

1997: Of the 154 entries for the second six months of 1997, 23 concerned buffer overflows, and 20 contained an exploit:

Table 3: Rootshell buffer overflow exploits from the second six months of 1997

Number	Date	OS	Architecture	Shellcode	Type	Restricted	Non-exec	Content	Special	Description
36	07/17/97	L	x86	LS	LZ					'splitvt'
37	07/19/97	L	x86	RS	LZ					NSCA 'httpd'
38	07/21/97	L	x86	LS	LZ					LD_PRELOAD
39	07/21/97	A	RS	LS	LZ					'xlock'
40	07/21/97	A	RS	LS	LZ					'lchangelv'
41	08/01/97	L	x86	RS	LZ					'innd'
42	08/03/97	A	RS	LS	LZ					'ping', 'in.rlogind'
43	08/05/97	FB	x86	LS	LZ					'xterm'
44	08/11/97	L, B	x86	LS	LZ		X		F	'lpr'
45	08/11/97	L	x86	LS	LZ					'color_xterm'
46	08/17/97	L	x86	LS	LZ				G	'traceroute', 'rsh', 'su', 'elm', 'ping', 'filter', 'minicom'
47	09/07/97	S	S	?	LZ					'eject'
48	09/25/97	L	x86	X	LZ					'samba'
49	09/26/97	L	x86	RS	LZ				H	'imapd'
50	10/30/97	B	x86	RS	LZ					'termcap'
51	10/30/97	L	x86	RX	LZ					'count.cgi'
52	10/30/97	B	x86	LS	LZ			ASM		'xterm_color'
53	11/14/97	L	x86	LS	LZ					'sperl'
54	11/14/97	L	x86	LS	LZ					Ideafix 'wm'
55	12/23/97	L	x86	LS	LZ				I	'su'

Special entries:

F: 44: Three exploits, one with guidelines to avoid a non-executable stack
G: 46: All exploits use the NLSPATH environment variable in the overflowing
H: 49: Contains a scanner looking for vulnerable machines, and then exploits them.
I: 55: Describes the vulnerable library function vsyslog() and demonstrates it with 'su'.

1998: Of the 181 entries for the first six months of 1998, 38 concerned buffer overflows, and 21 contained an exploit:

Table 4: Rootshell buffer overflow exploits from the first six months of 1998

Number	Date	OS	Architecture	Shellcode	Type	Restricted	Non-exec	Content	Special	Description
56	01/01/98	L	x86	RX	LZ					'count.cgi'
57	01/01/98	L	x86	LS	LZ					'traceroute'
58	01/06/98	S	S	?	LZ					'ping'
59	01/06/98	B	x86	LS	LZ					'sliplogin'
60	02/04/98	L	x86	LC	LZ				J	'ld.so', 'ld-linux.so'
61	02/19/98	L	x86	LS	LZ					'yapp'
62	02/19/98	L	x86	LC			X		K	'X'
63	02/19/98	L	x86	LS	LZ					'XFree86'
64	04/20/98	L	x86	LS	LZ				L	'lprm'
65	04/23/98	S	x86	LS	LZ				M	'ufsdump'
66	04/29/98	OB, FB	x86	LS	LZ			ASM		'lprm'
67	05/04/98	L	x86	LS	LZ			ASM		'xterm'
68	05/07/98	L	x86	LS					N	'dip'
69	05/13/98	L	x86	LS	LZ					'msgchk'
70	05/18/98	L	x86	RX	LZ				P	'count.cgi'
71	05/28/98	L	x86	LS	LZ					'xosview'
72	05/31/98	FB, L	x86	O	LZ				P	'named'
73	06/12/98	S	S	LS	LZ					'ufsrestore'
74	06/25/98	L	x86		LZ					'mailx'
75	06/29/98	L	x86	RS	LZ					'qpopper'
76	06/29/98	L	x86				X	C		'xterm'

Special entries:

J: 60: Overflow with a race condition. The overflow condition holds good only for a small time window.

K: 62: Guide for avoiding a non-executable stack by means of a shellcode run from the data segment, or exploiting with a modified stack frame in order to make the desired system calls.

L: 64: Very small shellcode, landing-zone with 'inc ecx'.

M: 65: Landing-zone with 'inc ecx'.

N: 68: Exact entry-point without a landing zone.

O: 72: The shellcode creates a remote shell by redirecting stdin/stdout/stderr to a socket.

1998: Of the 112 entries for the second six months of 1998, 22 concerned buffer overflows, and 9 contained an exploit:

Table 5: Rootshell buffer overflow exploits from the second six months of 1998

Number	Date	OS	Architecture	Shellcode	Type	Restricted	Non-exec	Content	Special	Description
77	07/08/98	W	x86			X		ASM	P	'SLMail'
78	07/09/98	S	S		LZ					'sendmail'
79	07/13/98	L	x86	LC	LZ					'crond'
80	07/17/98	L, FB, B	x86	RS	LZ			ASM	Q	'imapd'
81	07/18/98	L	x86		LZ			ASM	R	
82	09/13/98	L	x86	LC	LZ			ASM		'bash'
83	10/05/98	S, I, H		RC	LZ					'rpc.ttdbserver'
84	10/21/98	L	x86	RC	LZ			ASM	S	'netscape'
85	12/29/98	SCO	x86	RC	LZ			ASM		'calserver'

Special entries:

P: 77: Guide on creating an encoded alphanumeric shellcode and an egg that uses code from the environment for initiation. No landing zone required.

Q: 80: Several target operating systems, and with an advisory note.

R: 81: Shellcode variants: changing permissions for files, creating files like 'hosts.equiv', changing the host name and rebooting the computer.

S: 84: CGI script that sends the exploit from a malicious server to visiting browsers.

1999: At the time of this investigation there were entries for only two months in 1999, February and June. Of these 90 entries, 22 concerned buffer overflows, and 10 contained an exploit.

Table 6: Rootshell buffer overflow exploits from 1999

Number	Date	OS	Architecture	Shellcode	Type	Restricted	Non-exec	Content	Special	Description
86	02/10/99	D	A	LC	LZ			ASM	T	'at', 'inc'
87	02/10/99	L	X86	LS	LZ					'lpc'
88	02/15/99	W	X86		LZ				U	Mail-Max SMTP server
89	02/22/99	L	X86	LS	LZ			ASM		'lsof'
90	06/02/99	L	X86	RS	LZ			ASM		'ipop2d'
91	06/02/99	S	S	LS	LZ					'sdtcm_convert'
92	06/03/99	S	x86	LS	LZ					'lpset'
93	06/03/99	S	x86	LS	LZ					'admintool'
94	06/03/99	S	x86	LS	LZ					'dtprintinfo'
95	06/03/99	L	x86				X	ASM	V	'wu-ftp'

Special entries:

T: 86: Guide for exploiting Digital Unix after removing the non-executable stack. Generic exploit egg program directed against several overflows.
U: 88: Shellcode remotely downloads and executes a file.
V: 95: Prepares the stack to execute the desired system calls, and thus also works against a non-executable stack.

Origins of the Exploit Eggs

As the exploits rarely contained details on the author and the media from which they originated, the articles were taken as a starting point for studying the origins of the exploit eggs and the contained shellcode. Examination of the entries in Rootshell indicated that the following entries contained a shellcode from the article by Aleph One: #19, #35, #38, #45, #53, #57, #61, #62, #68, #69, #71, #74, #87 and #89. The exact shellcodes given by Mudge(1995) and Dildog (1998) were apparently not used, though their exploitation methods may have been employed. The shellcode for AIX presented by Smith (1997) was probably taken from one of the following entries: #17, #20, #21, #29, #39, #40 and #42, but the shellcode for HP-UX as presented by Smith was not found in any of the entries.

Types of Exploit Egg

Several of the overflow exploits were formed according to the guidelines presented by Aleph One (1996), or even using his exact exploit eggs and shellcode. Some generic features noted in the exploits are presented here.

A very common way of initiating the exploit is to overflow the buffer so that the return instruction pointer is overwritten with a prepared value pointing to the stack itself but below the current stack pointer. As the stack-frame does not always start from the same point of memory, a common way to be sure that the exploit points at the usable address is to prepare a 'landing zone' or 'drop zone' that contains only instructions that do something completely irrelevant to the operation of the shellcode. A useful instruction for that purpose is NOP (No Operation), but there are other possibilities as shown in entries #64 and #65. Cases using a landing zone are marked with (LZ). Of the 95 entries, 89 used this initializing method. An interesting case of a traditional overflow is entry #60, which involves an overflow with a race condition that requires exploitation to take place inside a time window. More details on the traditional types of exploits can be found in the article by Aleph One (1996) and in the notes published by Mudge (1995).

Some of the remaining 6 entries represented more complex methods: Entries #62, #76 and #95 avoid the non-executable stack protection by preparing the stack with parameters for making the wanted system calls. Entry #62 demonstrates another approach by running the shellcode from the data segment. #68 is a simple stack-based overflow that uses an exact address to jump to, probably due to the fixed address of the stack frame. #77 explains a method of encoding, and a way of initiating the exploit by using code from the

process space, a method presented by Dildog (1998), but it does not provide the exact exploit. The only heap-based overflow in the archive is the entry #30, which overflows data structures related to the IO stream stored in the heap.

Most of the exploit eggs of the traditional type contained a prepared shellcode, a small set of assembly instructions that is eventually executed. This set usually performs the instructions on the local or remote system with the privileges of the target software. All of the cases using the 'landing zone' approach had various kinds of shellcode to be run. In addition, #62 shows that the shellcode can be written into a data segment that is not protected by a non-executable stack.

The most common set of functions that are called up from the shellcode for Unix platforms were found to be:

- setuid(UID) [optional]
- seteuid(UID) [optional]
- setgid(GID) [optional]
- execve("/bin/sh", "sh", 0) [sometimes with parameter '-c' for running other programs]

The exploit string will usually contain no null characters, as, depending on the vulnerable string manipulation function, they will terminate a string in the C programming language. Sometimes the set of assembly operations that can be used in the exploit can be even further restricted. In entry #77 we see an exploitation method in which only alphanumeric codes can be used in the overflow string.

Exploits and shellcodes were available for most known operating systems. Notes were also available for overflowing a Macintosh operating system, but no exploit with functionality, thus the entry was not included in this material. Figures on the different operating systems exploited, with the processor architectures covered, are given below:

Win32: Covers Windows NT, 95 and 98 operating systems running on x86 processor architecture. Exploits: #77, #88

Solaris and SunOS: These usually apply to a Sparc architecture unless otherwise mentioned. Exploits: #5, #6, #9, #10, #11, #30, #47, #59, #65, #73, #78, #83, #91, #92, #93, #94

Linux: This seems to be the most popular system to write exploits against. The processor architecture is x86 unless otherwise mentioned. Exploits: #1, #2, #3, #13, #15, #16, #18, #19, #32, #33, #34, #35, #36, #37, #38, #41, #44, #45, #46, #48, #49, #51, #53, #54, #55, #56, #57, #60, #61, #62, #63, #64, #67, #68, #69, #70, #71, #72, #74, #75, #76, #79, #80, #81, #82, #84, #87, #89, #90, #95.

BSD: Includes at least OpenBSD, FreeBSD and BSDI on x86 architectures: #2, #4, #7, #8, #12, #14, #15, #43, #50, #52, #59, #66, #72, #80

HP-UX: Probably on PA-RISC architecture. Exploits: #83

IRIX: On MIPS architecture. Exploits: #22, #23, #24, #25, #26, #27, #28, #31

AIX: Probably on RS6000 architecture. Exploits: #17, #20, #21, #29, #39, #40, #42

SCO Unix: Probably on x86 architecture. Exploits: #85

Digital Unix: Version 4.0 of the operating system removed the non-executable stack protection by default (Granquist, 1999), and thus a generic exploit was created in #86. The shellcode is probably for Alpha architecture.

Discussion and Conclusions

www.rootshell.com

Since the www.rootshell.com archive did not appear to be updated for the current year 1999, several of the exploits currently available were not included in this material. On the other hand, 1998 was well covered and organized, and entries existed for the whole of 1997, though they were not so well organized. More detailed research could be considered for 1998 and 1997. The entries for 1996 and 1995 were mostly concerned with other vulnerabilities rather than overflows. Several entries consisted of exploits from more than one source, and some of the exploits have since been re-published or updated. The original source of the exploit was in most cases difficult to determine. As older exploits may be missing from this material, another archive should be studied to obtain more exploits from the time before 1996.

Figuring Out the Exploit Eggs

Several of the exploits may have been concealed in some form of packaging, e.g. in exploit collections or 'rootkits', and therefore escaped our attention. Of those studied here, it appears that the old methods described in the article by Aleph One (1996) seem still to be valid, as the methods of exploiting overflows remain the same. Some operating systems support a non-executable stack or some other means of protection against overflows, and thus more complex exploits have emerged.

A boom appears to have started immediately after the publication of the articles giving instructions on how to make your own exploit egg, as about 60 entries on overflows per year were stored in the database for 1997 and 1998. Almost all the entries for 1997 contained a working exploit, as opposed to only half of those for 1998, and the same low rate continued into the current year, of which only two months had been updated. This may be due to the fact that an exploit is so easy to create that if someone needs one, he can create one. The boom in creations and in public announcements of exploits appears to have faded, or the more skilled people have switched to other interests and the current status of expertise has been reduced to 'thousands of monkeys' trying out different inputs to programs in order to achieve their 15 minutes of fame.

Most exploits use similar methods. The operation of the shellcode usually varies only slightly, but the most common form of demonstrating the vulnerability still appears to be to create a remote or local shell for typing out the commands that imply elevated access rights. Some variants on the active content can be seen, but still nothing very large or complex appears in the archives. As just about any assembly instructions can be run on

the shellcode, there is little limitation as to what the intruder can do when given a way of inserting the malicious code.

On the Unix platform, several of the exploits give another UID besides 'root', but that does not slow down the writers of exploits. A remote shell with any UID, even that of 'nobody', provides access to the machine, and there are usually ways of elevating the privileges further.

Several methods are presented in the exploits converting the string containing the malicious code into something that the target software will accept. Through such encoding the shellcode can escape the filtering performed by the subject software, allowing the code to go through unchanged or at least still in an operational form.

Of the platforms covered here, the Linux operating system had the largest selection of exploits available: 50 entries. This may be due to that fact that it is a popular environment for people interested in hacking computer systems or securing them. Somewhat less entries were available for other systems, the Solaris and SunOS operating systems having 16 available and BSD variants for an Intel platform 14 entries. Of the other Unix variants, IRIX had 9 entries available, AIX 7, HP-UX 1, SCO Unix 1 and Digital Unix 1. The Win32 environment was a newcomer in the field, with Windows NT and 95/98 gaining in popularity as networked operating systems, with 2 entries available. These figures do not necessarily give any indication of the security of the operating systems, but may be due to the number of computer security people using them. The availability of an operating system for home use and the availability of the source code probably affect the numbers most. Also, these figures do not represent the number of exploits for each platform, as each entry can contain several vulnerabilities and exploits.

In addition to the ones already mentioned, several other guides to exploiting buffer overflows were included in the commentary for the exploit. With good documentation and assembly code, the exploit code itself can be worth reading for people who know assembly language. Sometimes the operation of the shellcode is also very well explained, if it is not so clear otherwise. For almost all of the operating systems mentioned, there was some kind of 'how to do it' guide available giving details of the stack structure.

Special Exploit Eggs

Sophisticated eggs have been provided when a straightforward egg has not been applicable for exploitation of the overflow in question. Discoverers have devised clever techniques for circumventing non-executable stacks and for privilege elevation by means of heap overflows.

Preparing a customized stack frame with no executable code in it has been shown to be a feasible means of penetration. The overflow area of the stack frame is crafted to contain addresses of desired function calls or system calls and their arguments, or addresses of chosen sets of machine instructions or strings embedded in memory-resident library code.

A correct sequence of addresses and arguments will result in the execution of a security-compromising instruction stream.

Stack frames have been manipulated to choose new downstream library calls in the host program in order to relocate the parasite code, and a new position can be chosen so that non-executable memory protections will be by-passed.

Overflow conditions have been shown to exhibit time window restrictions, narrowing down the time available for the exploitation, but an exploit for winning the race condition in question has been provided.

No reason to downplay the seriousness of heap overflows has been left after successful demonstrations of how they can be exploited. Compromises have been achieved by methods such as overwriting structures maintained by IO-related library calls and the procedure lookup table kept by the dynamic linker.

Some exploits use simple encoding methods to either disguise the commands used or avoid characters that would be filtered out or altered before the content was executed.

Generic execution programs were also available, in which the exploiting program takes the target software in as a parameter together with other exploitation parameters such as the command to be executed on the elevated privileges. The exploit egg is then generated, and submitted to the program. With improvements, this would make it easy for anyone to use more complex methods.

Where Can and Will the Overflows Roam

As people make mistakes, overflow vulnerabilities are probably here to stay. This particular vulnerability, like all implementation level vulnerabilities, is a result of a human error in the programming phase of the development, usually just the use of an insecure function instead of a more secure one. The mistake can appear in any module of any system, and affect the whole system around the insecure component.

It appears that most operating systems currently in use, such as all Unix variants and most Windows versions, are vulnerable to buffer overflow attacks. The problem is that the programmer who creates the software component is using easy but insecure methods to implement memory arrays and variables. The best solution would appear to improve software quality and the security awareness of the people creating the applications. Information on programming errors that cause security vulnerabilities seems to accumulate in the wrong places. The people who create the software remain in a cloud or switch to other tasks before learning enough, whereas the underground community stays interested n the field.

Spotting Vulnerable Programs

The security of software or of a system could be thoughtfully specified and designed according to the specification, but even so a small mistake in the programming phase, or a bad programming style, can create a hole that can be used to access the internal and more critical systems. Any program that created with C that uses C libraries might be using vulnerable functions.

When inspecting a system for vulnerable components and vulnerable points in the software, one way to start is by inspecting all possible interfaces with the outside world. All networked software at least should be under very thorough scrutiny. If the software does not pass the quality level needed, it should be disbanded from a security-critical system. In an operating system which is to be accessed by several users, at least all software modules that operate on elevated privileges and that the users can access should be examined or their use restricted. This policy of minimum access has been the normal attitude taken to security for a long time. One other underestimated field is the security of client software components. Any client module accessing or accepting connections or content from non-trusted servers or other clients can be open to implementation error vulnerabilities such as exploitable buffer overflows.

Securing the System

The simple addition of protection methods such as non-executable stack would prevent the most typical exploits from working, but this would not solve the problem of buffer overflows. Several methods are publicly known for circumventing this solution. Various means of validating the integrity of the stack can be introduced as well, e.g. separate stacks for function return addresses, or verification values in between the stack frames of functions. Both of these can still be exploited, with similar methods as the heap overflows, by modifying the contents of the stack and changing the behavior of the program without changing the return address. Several other solutions are discussed in the paper of Cowan et al. (1998), and a discussion of attacks and example exploits directed against them have appeared on the Bugtraq mailing list. A work-around such as is represented by these methods is always reactive, adding neither robustness nor preventing a skillful attacker from exploiting the vulnerability, as opposed to the proactive methods of fixing the overflows themselves before software release, thus improving the quality of the vulnerable software itself.

Recommendations for Researchers

For the sake of future research, the security knowledge that has already accumulated in the underground should perhaps be understood. Buffer overflows and other implementation errors should never be underestimated, and as researchers do not always have the payload of protecting the assets of the software vendor, a neutral and truthful standpoint could be taken. Overflows are not a complex matter, but they are caused by some problems that are difficult to solve.

The overflow exploits are akin to viruses, in that they are often built with low-level languages and are sometimes difficult to understand for someone who is used to higher level languages. Still, designing and building a buffer overflow exploit requires a lot less work, as all the function call interfaces that the intercepted process has are all available to the shellcode of the exploit.

In order to create better quality software and thus secure systems against overflows, several methods could be considered:

1) Promote awareness and secure programming practices so that programmers will make fewer mistakes. Education of the people responsible for the errors before they are even made can prove to be very profitable for developers in the long run, as correcting errors found in the testing phase and after deployment can be costly.

2) Developing operating systems, libraries, compilers and programming languages that have no buffer overflow vulnerabilities or are resistant to them. This could be the ultimate goal in the long run.

3) Since the targets of point 2 are going to take a while to achieve, we will need easily applicable and available local work-arounds such as non-executable stacks, compiler modifications or possibly other methods. These could limit the possibilities for most simple attacks against currently deployed systems.

4) One goal could be to provide detached work-arounds for systems that cannot be modified for secure operations. These methods could be similar to traffic filters and intrusion detection systems and would affect only remote attacks, as the system itself is left in an insecure state. Also work on local wrappers around the executables themselves could be considered further.

5) One major contribution to the field would be the developing of testing methods that set a baseline for the developers of security software. The wider the coverage of such methods, the better the overall security of the system would be, as the weak links would be tested against at least the simplest of attacks. The problem with these is often the cost involved, making the methods available only for those who can afford to pay.

6) Both black-box and white-box testing methods for finding existing buffer overflow errors in current and legacy software still in use should be developed further. The testing methods could be made available to the public for evaluating proprietary systems and systems currently in use.

Remember that the research results will benefit both developers and end-users. Solutions for combating buffer overflow exploits should be easily adaptable to existing software and

cheap to deploy in existing systems. Regular users should also have access to some methods that provide protection against such intrusions, and the methods they can access should be easily deployable and simple, so that they will be able to grasp the idea and the importance of the issue.

Issues for the 21st Century

User awareness and the education of software developers to understand the security issues involved in implementations could bring about a change in the attitudes of programmers and an understanding of the seriousness of the subject. It is to be hoped that wider academic participation in subjects related to implementation-level vulnerabilities will emerge in the future.

Some development can be expected in methods of protecting existing systems from buffer overflow vulnerabilities and in the deployment of protection methods, but as none of the protection methods attack the cause of the vulnerability proactively, but only the end result, there will be problems with exploitable overflows for years to come.

Even higher level languages performing bounds checking, such as Java, can be surprisingly dependent on lower-level native code, and thus they are not a panacea. Vulnerabilities at the lower level can affect the security of secure components that depend on insecure ones such as system libraries and components.

Some development could also take place on the hostile side. More exploits will probably consist of bootstrapping techniques, as they can download any program and execute it. Thus a very small exploit can start a process that can do very complex things when executed. Also, even though it is not usually necessary, an exploit can consist of a very large, sophisticated autonomous parasite program that can continue and perform the operations of the attacked program, and the desired additional operations on the side. These could go undetected, as the process identifier and operation of the original program would be maintained.

Also, bearing the results of this paper in mind, databases will probably be an issue in exploiting overflows in the next millennium. There are people and groups who are probably keeping an eye on the available public archives and learning from them. This is usually a good thing. As the proprietary databases of vulnerabilities also consist of large quantities of exploitation details, the risks involved should be acknowledged. Proprietary databases with vulnerabilities that the relevant vendors do not know about should probably be very well protected, or even better, avoided. All vulnerability data should be reported to the relevant vendor or developer, so that the security holes can be plugged. If that is not done, then the security of products will perhaps never improve.

References

Aleph One. (1996). Smashing the stack for fun and profit. Phrack [On-line], 49: file 14. Available: http://www.phrack.com/archive.html.

Bishop, M. (1993). Teaching computer security. Paper published in the proceedings of the Ninth IFIP International Symposium on Computer Security, pp. 43-52. Available: http://seclab.cs.ucdavis.edu/projects/vulnerabilities/scriv/1993-ifipsec.ps.

Bishop, M. (1996). Classifying vulnerabilities. Presented at the Nineteenth National Information Systems Security Conference, Baltimore, MD. Available: http://seclab.cs.ucdavis.edu/~bishop/scriv/1996-nissc-vn.ps.

Bishop, M. (1997). Teaching computer security. Position paper for the Workshop on Education in Computer Security, Monterey, CA. Available: http://seclab.cs.ucdavis.edu/~bishop/scriv/1997-wecs.ps.

Conover, M. (1999). w00w00 on heap overflows. Announced in the Bugtraq mailing list in Jan 1999. Available: http://www.w00w00.org/files/articles/heaptut.txt.

Cowan, C., Pu, C., Hinton, H. (1998). Death, taxes, and imperfect software: Surviving the inevitable. Paper presented at the New Security Paradigms Workshop 1998. Available: http://www.cse.ogi.edu/DISC/projects/immunix/bugtol.ps.gz.

Crvelin, H. (1999). Security Bugware List Page. Hacking information. Available: http://oliver.efri.hr/~crv/security/bugs/list.html.

Dildog. (1998). The Tao of Windows Buffer Overflow. Announced in the Bugtraq mailing list in Apr 1998. Available: http://www.cultdeadcow.com/cDc_files/cDc-351/.

eEye Digital Security Team. (1999a). Retina - Beta 1 Release notes. Available: http://www.eeye.com/Retina/beta1.html.

eEye Digital Security Team. (1999b). Retina vs. IIS4, Round 2 - The Exploit. Available: http://www.eeye.com/database/advisories/ad06081999/ad06081999-exploit.html.

Fyodor. (1998). Exploit world! Vulnerability information. Available: http://insecure.org/sploits.html

Garfinkel, S. Spafford, (1996) G. Practical Unix and Internet security. O'Reilly & Associates, Inc.

Ghosh, A.K., Schmid, M. & Shah, V. (1998). Testing the robustness of windows NT software. Experience report in the International Symposium on Software Reliability Engineering, Paderborn, GE. Available: ftp://ftp.rstcorp.com/pub/papers/issre98_CR.ps.

Granquist, L. (1999). Digital Unix 4.0 exploitable buffer overflows. From the Bugtraq mailinglist. Available: http://www.securityfocus.com/templates/archive.pike?list=1&date=1999-01-25

Herold, K.C. (1998). DOVES Abstract. In the proceedings of the 1998 UC Davis Student Workshop on Computing. Available: ftp://theory.cs.ucdavis.edu/swc98/abstracts/herold.ps.

Krsul, I. (1998a). Software vulnerability analysis. Department of Computer Sciences, Purdue University; PHD Thesis; Coast TR 98-09. Available: ftp://coast.cs.purdue.edu/pub/COAST/papers/ivan-krsul/krsul-phd-thesis.ps.Z.

Krsul, I. (1998b). Coast vulnerability database user's manual. Department of Computer Sciences, Purdue University. Coast TR 98-08. Available: ftp://coast.cs.purdue.edu/pub/COAST/papers/ivan-krsul/krsul-vdb-manual.ps.

L0pht Heavy Industries. (1999). SLINT. Software. Available: http://www.l0pht.com/slint.html

Laakso, M., Takanen, A. & Röning, J. (1999a). Runtime Symbol Interposition - Infiltrating the Black-box. In the proceedings of SANS'99 - The Eighth Annual Conference on System Administration, Networking and Security, Baltimore.

Laakso, M., Takanen, A. & Röning, J. (1999b). The vulnerability process: a tiger team approach to resolving vulnerability cases. In proceedings of the 11th FIRST Conference on Computer Security Incident Handling and Response. Brisbane. Available: http://www.ee.oulu.fi/research/ouspg/publications/FIRST1999-process/

Litchfield, D. (1999). Exploiting Windows NT 4 Buffer Overruns. Posted to Bugtraq mailing list in May 1999. Available: http://www.infowar.co.uk/mnemonix/ntbufferoverruns.htm.

Microsoft. (1999a). Patch Available for "Malformed Phonebook Entry" Vulnerability. Microsoft Security Bulletin (MS99-016). Available: http://www.microsoft.com/security/bulletins/ms99-016.asp

Microsoft. (1999b). "Access Violation" Error Message When You Quit Phone Dialer. Microsoft Knowledge Base ID: Q237185. Available: http://support.microsoft.com/support/kb/articles/q237/1/85.asp

Miller, B. P., Koski, D., Lee, C. P., Maganty, V., Murthy, R., Natarajan, A. & Steidl, J. (1995). Fuzz revisited: A re-examination of the reliability of UNIX utilities and services. CS-TR-95-1268. Available: ftp://grilled.cs.wisc.edu/technical_papers/fuzz-revisited.ps.Z.

Mudge. (1995). How to write Buffer Overflows. Available: http://l0pht.com/advisories/bufero.html.

Myers, J. (1993-1999). Bugtraq mailing list archives. Maintained since Oct'93. Available: http://www.geek-girl.com/bugtraq/index.html.

Netscape.org. (1995-1999). Archives of bugtraq@netspace.org. Maintained since June'95. Available: http://www.netspace.org/lsv-archive/bugtraq.html.

Packet Storm. (1999) Packet Storm Security Archives. Vulnerability information. Available: http://packetstorm.securify.com/archives.shtml

Rewted. (1999). Rewted network security labs: exploits. Vulnerability information. Available: http://www.rewted.org/exploits.

Rootshell. (1999). Welcome to Rootshell. Vulnerability information. Available: http://www.rootshell.com.

SecurityFocus. (1999). SecurityFocus. Vulnerability information. Available: http://www.securityfocus.com/

Smith, N.P. (1997). Stack smashing vulnerabilities in the UNIX operating system. Southern Connecticut State University. Available: http://destroy.net/machines/security/

Sommerfeld, B. (1997) Re: Smashing the stack. From the Bugtraq mailinglist. Available: http://www.securityfocus.com/templates/archive.pike?list=1&date=1997-01-21

Spafford, E. H. (1988) The internet worm program: An analysis. ACM Computer Communication Review; 19(1), pp. 17-57. Available: http://www.cs.purdue.edu/homes/spaf/tech-reps/823.ps.

UC Davis Vulnerabilities Project. (1999). DOVES. Available: http://seclab.cs.ucdavis.edu/projects/vulnerabilities/doves/index.html.

Voas, J. M. & McGraw, G. (1998). Software fault injection. John Wiley & Sons.