ATT vs Intel?!?

Almost everything in the Intel world of assembler is dealt with via Intel format assembler code. This is not how it is in the Unix world. Since the old PDP's the unix environment assembler syntax has followed ATT style. This might throw you off if you are used to any low level programming in the PC world. I would hate to see simple semantics discourage anyone so here are the main differences in a nutshell: [Figure 6].

I see where you're going but is this really as big as you say?

Many people just don't see how prevalent bounds checking problems are. The number of incidents in the Unix world should be proof enough. Sure there are a lot of false positives when you start going through source code but there are still many more areas where the buffer overflow can be a valid security concern. Finally programmers have started to drop the blatantly bad practices that most of the old well known coding problem were. Face it, people are slow and you have to hammer into their head what good coding practices are. Better yet they need to understand the theory and thinking of how hackers think and work in order to avoid most of the potential holes.

Although buffer overflow potential is a major problem in Unix 'C' programs... I proffer, from background and interaction, that this sort of problem is even MORE prevalent in the Microsoft world. When you find one of these in Windows 3.1/DOS or Win95 you pretty much own the barn as there isn't a really clear cut design of where rings 1,2,3,etc. live (i.e. everything is at ring 1 for all intents and purposes).

[example of number of stupid problems i.e. system(), moderate problems, getcwd(), and difficult problems - sprintf, etc. for a large package].

This is all that you really need to understand in order to further research buffer overflows and, with a text book or a friend, start writing your own exploits. Programmers, are you getting this?!?! How much more poor coding must people endure before you've riddled everything in all of the major operating systems with low level bugs and holes!

Details! I want details!

The Intel stuff is fairly straight forward. If you get stuck all you need to do is consult your local virus writer. He/she will be able to give you plenty of details. So... since you probably have a good contact for Intel OS's (FreeBSD, BSDI, Linux, etc.) I'll do the details on the SPARC setup. [What's that you say? You don't know any good viral writers? Shame on you! These people will be able to open up an entire world of exploits that certain groups have enjoyed singular possession of.]

Here's what a stack frame in Solaris looks like: [figure 4].

[side note... the callee... not the caller, has to shift the register window and adjust the stack pointer in the SPARC architecture.. not the caller]

Understanding that this is the information on the stack (as referenced by the stack pointer) you should be able to see that if you overwrite the instruction pointer with an address of your preference and let the routine do it's RET, you will start executing whatever code you want.

All you need to do is something the equivalent of:

```
for (i=0; i< 4096; i++)

buffer[i] = 0x90;
```

(where buffer is really something like: char buffer[2];) This will start trampling over things fairly quickly.

Using the above pseudo example and a program like gdb you will quickly see where you need to be overwriting.

What does the code I have the IP point to need to look like?

[figure 5 - libc problem]

Great... what's so cool about this (aka... I don't get it)?

To understand the importance of this it is necessary to understand a little about the structure of a 'C' program when it is run and also a little about how the processor deals with the machine code beneath it [figure 1].

Next one needs to understand how the underlying architecture deals with the Instruction Pointer (often referred to as the Program Counter), a little about the registers on the chip in question and what they reference [figure 2].

The IP register points to either the address of the next instruction to be executed or the address of the instruction currently being executed (depending upon how the designers set things up). This is the crux of the matter at hand. In general terms, the coder does not directly access the IP register. After each instruction is executed the IP value is automatically incremented to point to the address of the next instruction [figure 3].

Now, when a call is made in your program the system needs to know where to go for the next instruction and how to get back to the last place it was. The call instruction usually specifies the value needed to be added to the IP for the address of the next instruction to be executed and pushes the current IP onto the stack (this is oversimplified as there are nuances between how different architectures and systems deal with this... if anyone has questions they can ask me after the conference over a beer). The return instruction in the called function pops the stack value back into the IP to resume execution at the next instruction after the call.

Yeah... so? I'm getting bored...

You remember the stack in figure 1 don't you? This is where this information is being stored and retrieved from. If someone doesn't do correct bounds checking you can write all the way from the heap, through the unused address space (if it exists on this architecture) and into the stack. Heck, you can even write through the stack if you want and have fun with the command line args and environment variables.

All you need to do is be clever enough to overwrite the saved IP that is on the stack with the location that you want to IP to point to upon return. This address will presumably contain the opcodes and operands of the code that you have constructed and put at this address. Perhaps something ingenious like execve("/bin/sh", 0, 0) / syscall(59,"/bin/sh", 0, 0); or even something nasty like the machine instruction for HLT (assuming you are in the proper ring).

Compromised - Buffer-Overflows, from Intel to SPARC Version 8

mudge@l0pht.com

The purpose of this talk is to familiarize people with buffer overflows. What they are, why they work and how to approach them.

What the heck is a buffer overflow?

A buffer overflow occurs when an object of size x+y is placed into a container of size x. This can happen in many situations when the programmer does not take proper care to bounds check what their functions do and what they are placing into variables inside their programs. As usual, the fun begins when this programming mistake is made at a place that allows user definable data to be inserted.

Some common examples are:

(less common)

```
char input[20];

gets(input);
```

(more common)

```
char env[20];

env = getenv("FOOBAR");
```

Buffer overflows are by far the most common security problem in coding. For every system(), popen(), etc. that you find in source code there are at least 20x's as many places where the potential for abuse through improper bounds checking exists. Thus is one of the great legacies that the 'C' programming language affords us. gets() and copying environments are by no means the only place for buffer overflows to happen.