“Beware that WinAmp skin

“The popular skinning feature in Nullsoft’s WinAmp media player has left the door wide open for malicious attackers to hijack PCs.

“Security researchers at K-Otik discovered the vulnerability and released details of a ‘Skinhead’ zero-day exploit that is already spreading in the wild. The exploit, which targets WinAmp versions 3.x and 5.x, is being used to forcefully install spyware and Trojans on infected systems.”
Assignment due 2004.08.25: read foreword and preface of textbook.

Assignment due today: read textbook Chapter 1 pages 1–14, up to “The Trinity of Trouble.”

Assignment due 2004.08.30: read the rest of Chapter 1.
Example:

```c
void zork(int a)
{
    int b;
    b = a + 5;
}

int main(int argc, char **argv)
{
    zork(3);
}
```
What typical computer actually does:

zork:

--sp;
sp[0] = sp[2] + 5;
++sp;
goto *sp++;

main:

*--sp = 3;
*--sp = t76;
goto zork;
t76: ++sp;
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>st+512</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>st+512</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>3</td>
<td>st+511</td>
</tr>
<tr>
<td>0</td>
<td>t76</td>
<td>3</td>
<td>st+510</td>
</tr>
<tr>
<td>0</td>
<td>t76</td>
<td>3</td>
<td>st+510</td>
</tr>
<tr>
<td>0</td>
<td>t76</td>
<td>3</td>
<td>st+510</td>
</tr>
<tr>
<td>0</td>
<td>t76</td>
<td>3</td>
<td>st+509</td>
</tr>
<tr>
<td>8</td>
<td>t76</td>
<td>3</td>
<td>st+509</td>
</tr>
<tr>
<td>8</td>
<td>t76</td>
<td>3</td>
<td>st+510</td>
</tr>
<tr>
<td>8</td>
<td>t76</td>
<td>3</td>
<td>st+511</td>
</tr>
<tr>
<td>8</td>
<td>t76</td>
<td>3</td>
<td>st+512</td>
</tr>
</tbody>
</table>

main:

*--sp = 3

*--sp = t76
goto zork

zork:

--sp

sp[0]=sp[2]+5

++sp
goto *sp++
t76:

++sp

st+512
More concise stack diagram:

main:
  *---sp = 3
  3 *---sp = t76
  t76 3 goto zork
  t76 3 zork:
  t76 3 --sp
  0 t76 3 sp[0]=sp[2]+5
  8 t76 3 ++sp
  t76 3 goto *sp++
  3 t76:
  3 ++sp

Diagram has blanks for sp[-1], sp[-2], etc., rather than actual memory contents.
Example:

```c
void *ptr;
void one(int a)
{  ptr = (&a)[-1];  }
void two(void)
{  one(7);  printf("two\n");  }
void three(int a)
{  (&a)[-1] = ptr;  }
void four(void)
{  three(9);  printf("four\n");  }
int main(int argc,char **argv)
{  two();  four();  }
```
What computer actually does:

```c
void *ptr;
one: ptr = sp[0]; goto *sp++;
two:
    *--sp = 7; *--sp = t38;
goto one; t38: ++sp;
    printf("two\n"); goto *sp++;
three: sp[0] = ptr; goto *sp++;
four:
    *--sp = 9; *--sp = t70;
goto three; t70: ++sp;
    printf("four\n"); goto *sp++;
main:
    *--sp = t130; goto two; t130:
    *--sp = t139; goto four;
t139: ;
```
<table>
<thead>
<tr>
<th>stack</th>
<th>ptr</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>*--sp = t130</td>
</tr>
<tr>
<td>t130</td>
<td>goto two</td>
</tr>
<tr>
<td>t130</td>
<td>two: *--sp = 7</td>
</tr>
<tr>
<td>7</td>
<td>*--sp = t38</td>
</tr>
<tr>
<td>t130</td>
<td>goto one</td>
</tr>
<tr>
<td>7</td>
<td>one: ptr = sp[0]</td>
</tr>
<tr>
<td>t38</td>
<td>goto *sp++</td>
</tr>
<tr>
<td>7</td>
<td>t38: ++sp; print two</td>
</tr>
<tr>
<td>t130</td>
<td>goto *sp++</td>
</tr>
<tr>
<td>t38</td>
<td>t130: *--sp = t139</td>
</tr>
<tr>
<td>t139</td>
<td>goto four</td>
</tr>
<tr>
<td>t38</td>
<td>four: *--sp = 9</td>
</tr>
<tr>
<td>9</td>
<td>*--sp = t70</td>
</tr>
<tr>
<td>t139</td>
<td>goto three</td>
</tr>
<tr>
<td>9</td>
<td>three: sp[0] = ptr</td>
</tr>
<tr>
<td>t38</td>
<td>goto *sp++</td>
</tr>
<tr>
<td>9</td>
<td>t38: ++sp; print two</td>
</tr>
<tr>
<td>t139</td>
<td>goto *sp++</td>
</tr>
</tbody>
</table>
Why two instead of four?
Answer: (&a)[-1] = ptr
changed the control flow
by changing three’s return address.

This behavior isn’t obvious!
And almost certainly isn’t desired.

Typical buffer-overflow attack
writes past end (or beginning) of an array
to change a return address.
The new return address
points to attacker’s instructions.

Examples later.
Let’s look inside a real computer.

Architecture: x86, i.e., 80386-compatible.
CPU: Pentium M.
Operating system: FreeBSD 4.10.
Compiler: gcc 2.95.4, with -fomit-frame-pointer option.

Some addresses in process memory:
&one is 0x80484a0.
&two is 0x80484ac.
&three is 0x80484d0.
&four is 0x80484dc.
&main is 0x8048500.
&ptr is 0x8049644.

(In gdb: disas one; print &ptr.)
Compiled instructions in memory:

0x80484a0 (one): \( ax = *sp \)

0x80484a3: \( *0x8049644 = ax \)

0x80484a8: \( \text{goto } *sp++ \)

0x80484ac (two): \( sp -= 3 \)

0x80484af: \( sp -= 3 \)

0x80484b2: \( *--sp = 7 \)

0x80484b4: \( *--sp = 0x80484b9 \)

0x80484b9 (t38): \( sp += 4 \)

0x80484bc: \( sp -= 3 \)

0x80484bf: \( *--sp = 0x804854b \)

0x80484c4: \( *--sp = 0x80484c9 \)

0x80484c4: \( \text{goto } 0x8048350 \)

... 

Note extra \( sp \) motion, used for alignment.
gdb-format x86 assembly language:
0x80484a0: mov (%esp,1),%eax
0x80484a3: mov %eax,0x8049644
0x80484a8: ret
0x80484ac: sub $0xc,%esp
0x80484af: add $0xfffffffff4,%esp
0x80484b2: push $0x7
0x80484b4: call 0x80484a0
0x80484b9: add $0x10,%esp
0x80484bc: add $0xfffffffff4,%esp
0x80484bf: push 0x804854b
0x80484c4: call 0x8048350

... 

Warning: Pointer arithmetic counts bytes in this language. Subtracting 12 from sp here is like sp -= 3 in C.
Actual bytes in memory:
0x80484a0: 8b 04 24
0x80484a3: a3 44 96 04 08
0x80484a8: c3
0x80484a9: 8d 76 00 (unused)
0x80484ac: 83 ec 0c
0x80484af: 83 c4 f4
0x80484b2: 6a 07
0x80484b4: e8 e7 ff ff ff
0x80484b9: 83 c4 10
0x80484bc: 83 c4 f4
0x80484bf: 68 4b 85 04 08
0x80484c4: e8 87 fe ff ff ff
...

Will learn about machine language later.