Ethical Hacking and Countermeasures

Version 6

Module XXIV

Buffer Overflows

February 4, 2008 10:51 AM PST

Facebook, MySpace image uploaders vulnerable to attack

Posted by Robert Vamosi

Updated at 3:37 p.m. PST with statement from MySpace and Facebook.

Within the last week, researcher Elazar Broad has disclosed two ActiveX vulnerabilities in the tools that MySpace.com and Facebook users use to upload images to their sites. On Sunday, Broad disclosed a buffer overflow vulnerability within the Facebook image upload control. Last week, Broad disclosed a similar buffer overflow flaw within MySpaceAurigma's ImageUploader ActiveX; the MySpace vulnerability also affects Facebook users.

Facebook and MySpace use controls repackaged from Aurigma Imaging Technology. Vulnerable to the recent attack scenario are FaceBook PhotoUploader 4.5.57.0, Aurigma ImageUploader4 4.6.17.0, Aurigma ImageUploader4 4.5.70.0, Aurigma ImageUploader4 4.5.126.0, and Aurigma ImageUploader5 5.0.10.0.

The MySpace attack outlined last week could allow specially crafted Web pages to crash Windows systems. The Facebook attack announced Sunday could allow for denial-of-service attacks or for malicious code to run on compromised PCs. An exploit exists for the MySpace attack. An exploit for the Facebook attack is expected to be posted on the Internet shortly.

Recent versions of Facebook PhotoUploader 4.5.57.1 are not vulnerable. Also, for the MySpace vulnerability, Aurigma Imaging Technology recommends upgrading to the latest 4.x and 5.x releases.

Source: *http://www.news.com/*

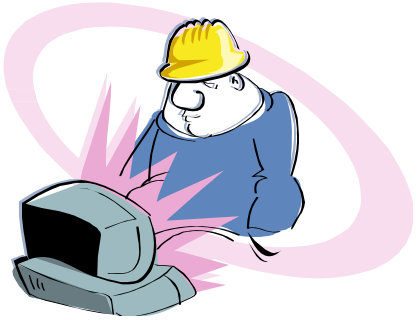
Scenario

It was a job that Tim wanted right from the start of his career. Being a Project Manager at a well-known software firm was definitely a sign of prestige. But now, his credibility was at stake.

The last project that Tim handled failed to deliver because the application crashed. The customer of Tim's company suffered a huge financial loss.

At the back of his mind, something was nagging Tim...

Had he asked his Test Engineers to do a thorough testing of the delivered package, this would not have happened.

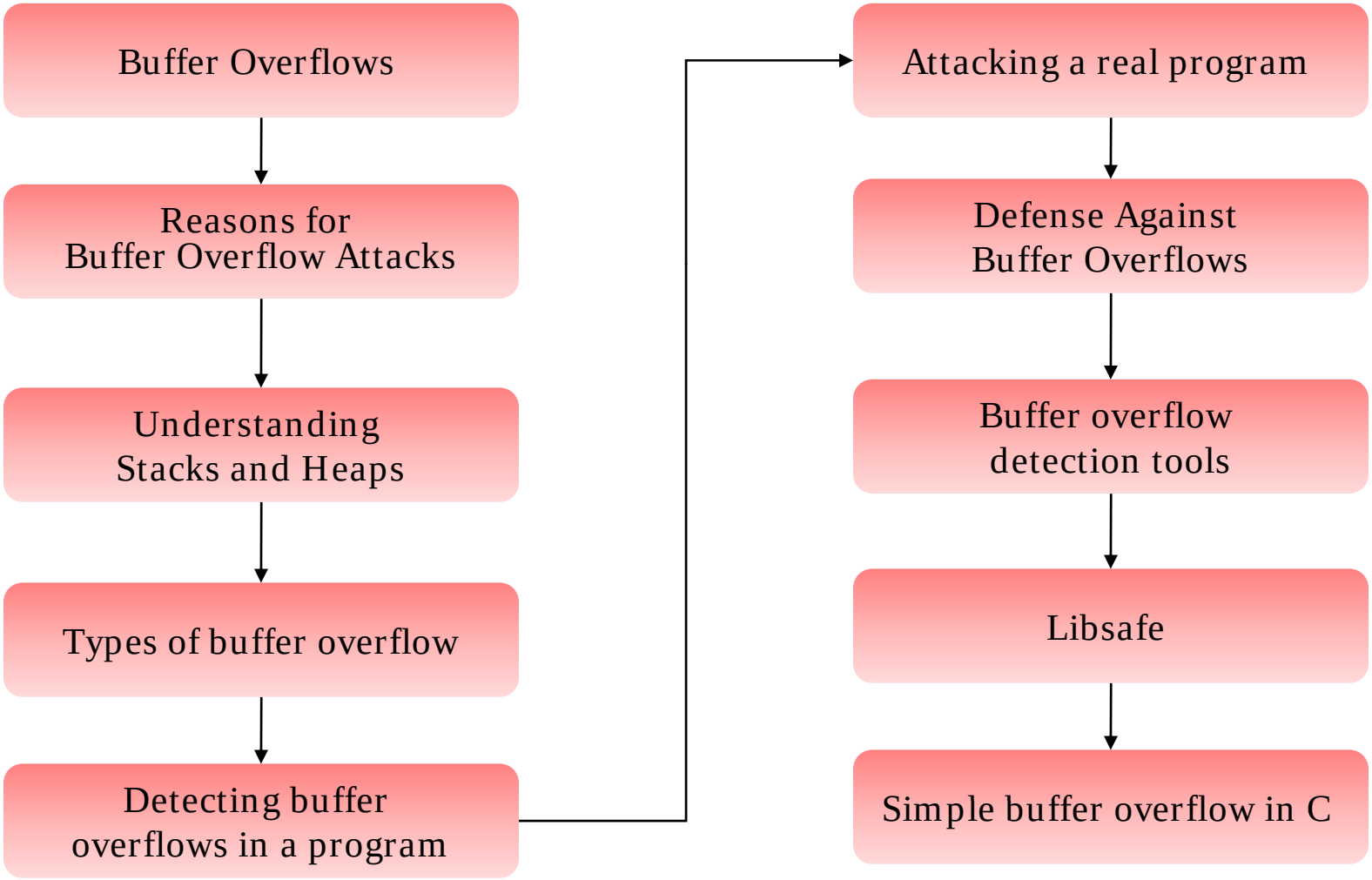


Module Objective

This module will familiarize you with :

- Buffer Overflows
- Reasons for buffer overflow attacks
- Understanding Stacks and Heaps
- Types of buffer overflow
- Detecting buffer overflows in a program
- Attacking a real program
- Defense Against Buffer Overflows
- Buffer overflow detection tools
- Libsafe
- Simple buffer overflow in C

Module Flow



Real World Scenario

heise Security — IT security news and services at heise Security UK

31 January 2008, 12:48

XnView vulnerable to a buffer overflow attack

Security service provider Secunia has on Wednesday reported that a buffer overflow can occur in the popular **XnView image viewer[1]** and converter when handling manipulated high dynamic range images in the Radiance RGBE (`.hdr`) format. When opened, specially crafted `.hdr` files can inject and execute a trojan via this bug. An update for the Windows version is available, but versions for other operating systems have yet to be patched.

The vulnerability is due to a flawed length check in versions 1.91 and 1.92 of XnView. The developer believes that previous versions also contain the flawed code. Versions for Windows, Mac OS X, Linux, BSD, Irix, Solaris, HP-UX and AIX are all probably affected. The vulnerability is also present in NConvert 4.85 and the GFL SDK 2.870 software development kit.

The developer has made the current versions 1.9 2.1 of XnView and 4.86 of NConvert for Windows available for downloading. There is no update yet for GFL-SDK or for Mac and UNIX versions. Users are advised to install the fixed version as soon as possible. Users of platforms for which no update is yet available should not open any `.hdr` files from unknown or suspect sources with the software.

Source: *http://www.heise-online.co.uk/*

Why are Programs/ Applications Vulnerable

Boundary checks are not done fully or, in most cases, they are skipped entirely

Programming languages, such as C, have errors in it

The strcat(), strcpy(), sprintf(), vsprintf(), bcopy(), gets(), and scanf() calls in C language can be exploited because these functions do not check to see if the buffer, allocated on the stack, is large enough for the data copied into the buffer

Programs/ applications are not adhered to good programming practices

Buffer Overflows

A generic buffer overflow occurs when a buffer that has been allocated a specific storage space, has more data copied to it than it can handle

Consider the following source code. When the source is compiled and turned into a program and the program is run, it will assign a block of memory 32 bytes long to hold the name string

```
#include<stdio.h>
     int main ( int argc , char **argv)
          {
               char target[5]="TTTT";
               char attacker[11]="AAAAAAAAAA";
               strcpy( attacker," DDDDDDDDDDDDD");
               printf("% \n",target);
               return 0;
          }
```

This type of vulnerability is prevalent in UNIX- and NT-based systems

Reasons for Buffer Overflow Attacks

Buffer overflow attacks depend on two things:

- The lack of boundary testing
- A machine that can execute a code that resides in the data/stack segment

The lack of boundary is common and, usually, the program ends with the segmentation fault or bus error

In order to exploit buffer overflow to gain access to or escalate privileges, the offender must create the data to be fed to the application

Random data will generate a segmentation fault or bus error, never a remote shell or the execution of a command

Knowledge Required to Program Buffer Overflow Exploits

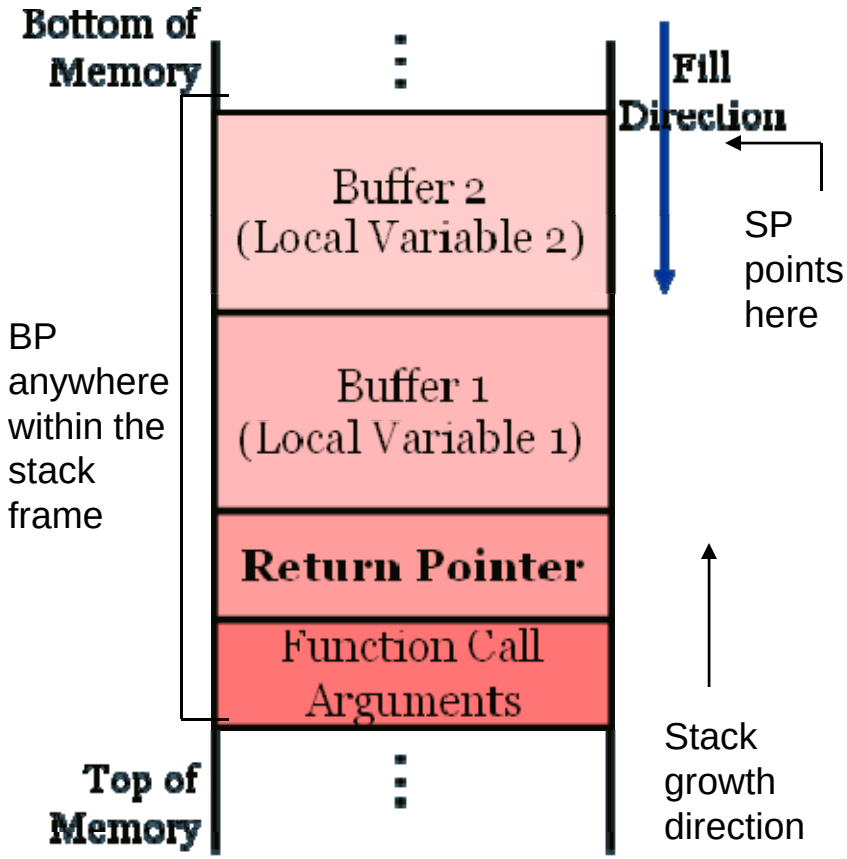
- C functions and the stack
- A little knowledge of assembly/ machine language
- How system calls are made (at the machine code level)
- exec( ) system calls
- How to guess some key parameters

Understanding Stacks

The stack is a (LIFO) mechanism that computers use to pass arguments to functions as well as to refer to the local variables

It acts like a buffer, holding all of the information that the function needs

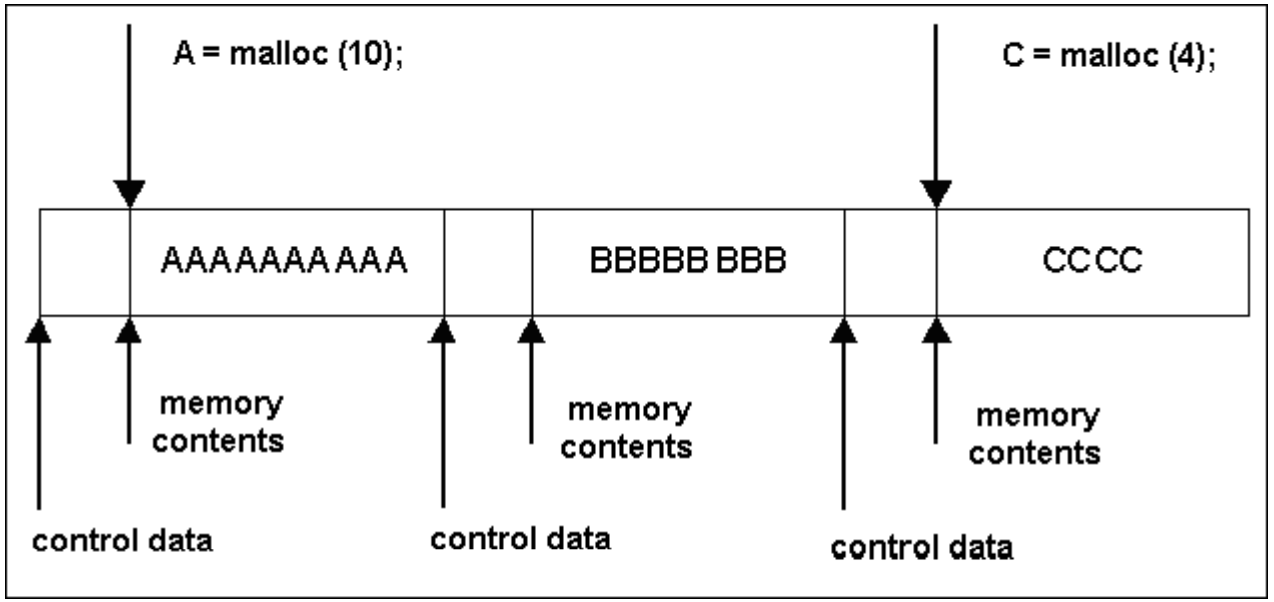
The stack is created at the beginning of a function and released at the end of it



Understanding Heaps

The heap is an area of memory utilized by an application and is allocated dynamically at the runtime

Static variables are stored on the stack along with the data allocated using the malloc interface



Simple Heap Contents

Types of Buffer Overflows: Stack-Based Buffer Overflow

A stack overflow occurs when a buffer has been overrun in the stack space

Malicious code can be pushed on the stack

The overflow can overwrite the return pointer so that the flow of control switches to the malicious code

C language and its derivatives offer many ways to put more data than anticipated into a buffer

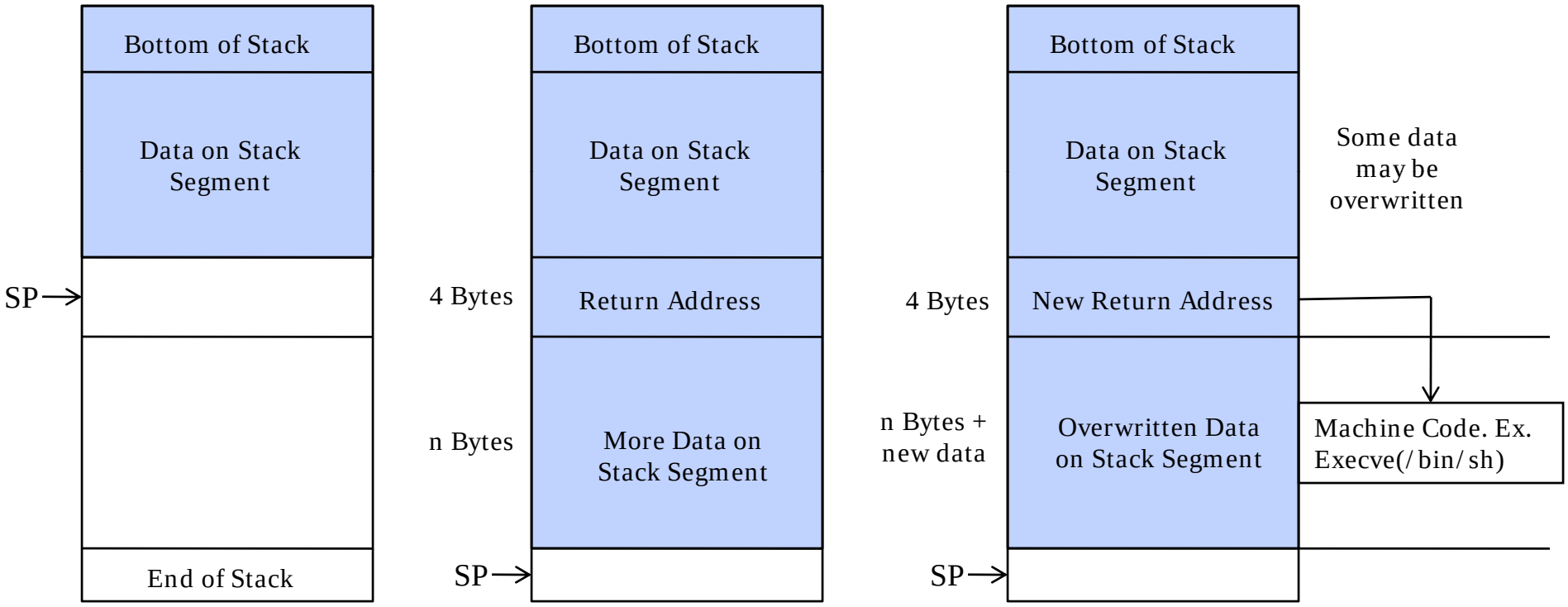
Consider an example program given on the next slide for simple uncontrolled overflow

- The program calls the bof() function
- Once in the bof() function, a string of 20 As is copied into a buffer that holds 8 bytes, resulting in a buffer overflow

A Simple Uncontrolled Overflow of the Stack

```
/* This is a program to show a simple uncontrolled overflow of the stack. It will
   overflow EIP with 0x41414141, which is AAAA in ASCII.  */
#include <stdlib.h>
#include <stdio.h>
#include <string.h>
int bof()
{
char buffer[8]; /* an 8 byte character buffer */
strcpy(buffer,"AAAAAAAAAAAAAAAAAAAA"); /*copy 20 bytes of A into the buffer*/
return 1; /*return, this will cause an access violation due to stack corruption.*/
}
int main(int argc, char **argv)
{
bof(); /*call our function*/
/*print a short message, execution will  never reach this point because of the
   overflow*/
printf("Lets Go\n");
return 1; /*leaves the main function*/
}
```

Stack Based Buffer Overflows



Types of Buffer Overflows: Heap-Based Buffer Overflow

Variables that are dynamically allocated with functions, such as malloc(), are created on the heap

In a heap-based buffer overflow attack, an attacker overflows a buffer that is placed on the lower part of heap, overwriting other dynamic variables, which can have unexpected and unwanted effects

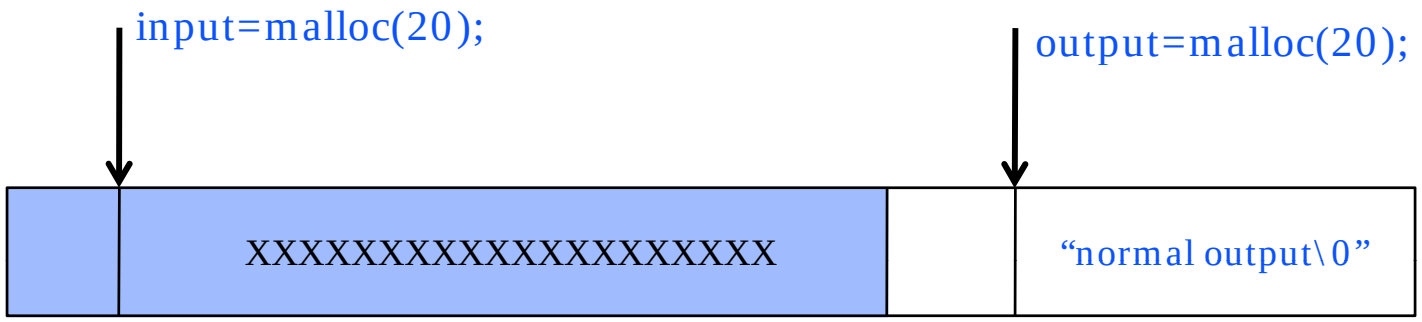
If an application copies data without first checking whether it fits into the target destination, the attacker could supply the application with a piece of data that is large, overwriting heap management information

In most environments, this may allow the attacker to control over the program's execution

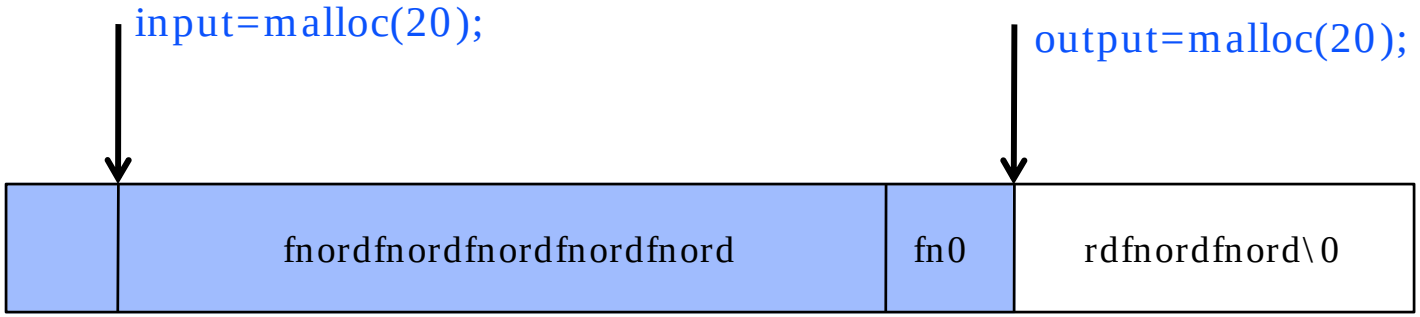
Heap Memory Buffer Overflow Bug

```
/*heap1.c – the simplest of heap overflows*/
#include <stdio.h>
#include <stdlib.h>

int main(int argc, char *argv[])
{

char *input = malloc (20);
char *output = malloc (20);

strcpy (output, "normal output");
strcpy (input, argv[1]);

printf ("input at %p: %s\n", input, input);
printf ("output at %p: %s\n", output, output);

printf("\n\n%s\n", output);
}
```

Heap-Based Buffer Overflow



Heap: Before Overflow



Heap: After Overflow

Understanding Assembly Language

The two most important operations in a stack:

- 1. Push – put one item on the top of the stack
- 2. Pop – "remove" one item from the top of the stack

Typically, returns the contents pointed to by a pointer and changes the pointer (not the memory contents)

- **EIP** The extended instruction pointer. This point to the code that you are currently executing. When you call a function, this gets saved on the stack for later use.
- **ESP** The extended stack pointer. This points to the current position on the stack and allows things to be added and removed from the stack using push and pop operations or direct stack pointer manipulations.
- **EBP** The extended base pointer. This register should stay the same throughout the lifetime of the function. It serves as a static point for referencing stack-based information like variables and data in a function using offsets. This almost always points to the top of the stack for a function.

Shellcode

Shellcode is a method to exploit stack-based overflows

Shellcodes exploit computer bugs in how the stack is handled

Buffers are soft targets for attackers as they overflow easily if the conditions match

```
"\x2d\x0b\xd8\x9a\xac\x15\xa1\x6e\x2f\x0b\xdc\xda\x90\x0b\x80\x0e"
  "\x92\x03\xa0\x08\x94\x1a\x80\x0a\x9c\x03\xa0\x10\xec\x3b\xbf\xf0"
  "\xdc\x23\xbf\xf8\xc0\x23\xbf\xfc\x82\x10\x20\x3b\xaa\x10\x3f\xff"
  "\x91\xd5\x60\x01\x90\x1b\xc0\x0f\x82\x10\x20\x01\x91\xd5\x60\x01"
```

How to Detect Buffer Overflows in a Program

There are two ways to detect buffer overflows:

One way is to look at the source code

- In this case, the hacker can look forstrings declared as local variables in functions or methods and verify the presence of boundary checks
- It is also necessary to check for improper use of standard functions, especially those related to strings and input/output

Another way is to feed the application with huge amounts of data and check for the abnormal behavior

Attacking a Real Program

Assuming that a string function is being exploited, the attacker can send a long string as the input

This string overflows the buffer and causes a segmentation error

The return pointer of the function is overwritten, and the attacker succeeds in altering the flow of execution

If the user has to insert his code in the input, he/she has to:

- Know the exact address on the stack
- Know the size of the stack
- Make the return pointer point to his code for execution

NOPS

- Most CPUs have a No Operation (NOP) instruction – it does nothing but advance the instruction pointer
- Usually, you can put some of these ahead of your program (in the string)
- As long as the new return address points to a NOP, it is OK
- Attacker pads the beginning of the intended buffer overflow with a long run of NOP instructions (a NOP slide or sled) so the CPU will do nothing until it gets to the 'main event' (which preceded the 'return pointer')
- Most intrusion detection systems (IDSs) look for signatures of NOP sleds
- ADMutate (by K2) accepts a buffer overflow exploit as input and randomly creates a functionally equivalent version (polymorphism)

How to Mutate a Buffer Overflow Exploit

For the NOP portion

- Randomly replace the NOPs with functionally equivalent segments of code (e.g.: x++; x-; ? NOP NOP)

For the "main event"

- Apply XOR to combine code with a random key unintelligible to IDS. The CPU code must also decode the gibberish in time in order to run the decoder. By itself, the decoder is polymorphic and, therefore, hard to spot

For the "return pointer"

- Randomly tweak LSB of pointer to land in the NOP-zone

Once the Stack is Smashed...

Once the vulnerable process is commandeered, the attacker has the same privileges as the process and can gain normal access. He/she can then exploit a local buffer overflow vulnerability to gain super-user access

Create a backdoor

- Using (UNIX-specific) inetd
- Using Trivial FTP (TFTP) included with Windows 2000 and some UNIX flavors

Use Netcat to make raw and interactive connections

- Shoot back an Xterminal connection
- UNIX-specific GUI

Defense Against Buffer Overflows

Manual auditing of code

Disabling stack execution

Safer C library support

Compiler techniques

Tool to Defend Buffer Overflow: Return Address Defender (RAD)

RAD is a simple patch for the compiler that automatically creates a safe area to store a copy of return addresses

After that, RAD automatically adds protection code into applications and compiles them to defend programs against buffer overflow attacks

RAD does not change the stack layout

Tool to Defend Buffer Overflow: StackGuard

StackGuard: Protects systems from stack smashing attacks

StackGuard is a compiler approach for defending programs and systems against "stack smashing" attacks

Programs that have been compiled with StackGuard are largely immune to stack smashing attacks

Protection requires no source code changes at all. When a vulnerability is exploited, StackGuard detects the attack in progress, raises an intrusion alert, and halts the victim's program

Tool to Defend Buffer Overflow: Immunix System

Immunix System is an Immunix-enabled RedHat Linux distribution and suite of the application-level security tools

Immunix secures a Linux OS and applications

Immunix works by hardening the existing software components and platforms so that attempts to exploit security vulnerabilities will fail safe

The compromised process halts instead of giving control to the attacker, and then is restarted

Vulnerability Search: NIST 1

Sponsored by DHS National Cyber Security Division/US-CERT

NIST National Institute of Standards and Technology

National Vulnerability Database

automating vulnerability management, security measurement, and compliance checking

Vulnerabilities | Checklists | Product Dictionary | Impact Metrics | Data Feeds | Statistics

Home | ISAP/SCAP | SCAP Validated Tools | SCAP Events | About | Contact | Vendor Comments

Mission and Overview

NVD is the U.S. government repository of standards based vulnerability management data. This data enables automation of vulnerability management, security measurement, and compliance (e.g. FISMA).

Resource Status

NVD contains:

29705 CVE Vulnerabilities
150 Checklists
132 US-CERT Alerts
2152 US-CERT Vuln Notes
3171 OVAL Queries
13723 Vulnerable Products

Last updated: 02/25/08
CVE Publication rate:

There are **1304** matching records. Displaying matches **1** through **20**.

Next 20 Matches

CVE-2008-0835

Summary: SQL injection vulnerability in indexen.php in Simple CMS 1.0.3 and earlier allows remote attackers to execute arbitrary SQL commands via the area parameter.
Published: 2/20/2008
CVSS Severity: 7.5 (High)

CVE-2008-0820

Summary: ** DISPUTED ** Cross-site scripting (XSS) vulnerability in index.php in Etomite 0.6.1.4 Final allows remote attackers to inject arbitrary web script or HTML via $_SERVER['PHP_INFO']. NOTE: the vendor disputes this issue in a followup, stating that the affected variable is $_SERVER['PHP_SELF'], and "This is not an Etomite specific exploit and I would like the report rescinded."
Published: 2/19/2008
CVSS Severity: 4.3 (Medium)

CVE-2008-0782

Summary: Directory traversal vulnerability in MoinMoin 1.5.8 and earlier allows remote attackers to overwrite arbitrary files via a .. (dot dot) in the MOIN_ID user ID in a cookie for a userform action. NOTE: this issue can be leveraged for PHP code execution via the quicklinks parameter.
Published: 2/14/2008

Vulnerability Search: NIST 2

National Cyber-Alert System

Vulnerability Summary CVE-2008-0835

Original release date: 2/20/2008
Last revised: 2/21/2008
Source: US-CERT/NIST

Overview

SQL injection vulnerability in indexen.php in Simple CMS 1.0.3 and earlier allows remote attackers to execute arbitrary SQL commands via the area parameter.

Impact

CVSS Severity (version 2.0):
CVSS v2 Base score: 7.5 (High) (AV:N/AC:L/Au:N/C:P/I:P/A:P) (legend)
Impact Subscore: 6.4
Exploitability Subscore: 10.0

Access Vector: Network exploitable
Access Complexity: Low
Authentication: Not required to exploit
Impact Type: Provides user account access, Allows partial confidentiality, integrity, and availability violation , Allows unauthorized disclosure of information , Allows disruption of service

References to Advisories, Solutions, and Tools

External Source: BID (disclaimer)
Name: 27843
Hyperlink: http://www.securityfocus.com/bid/27843

Valgrind

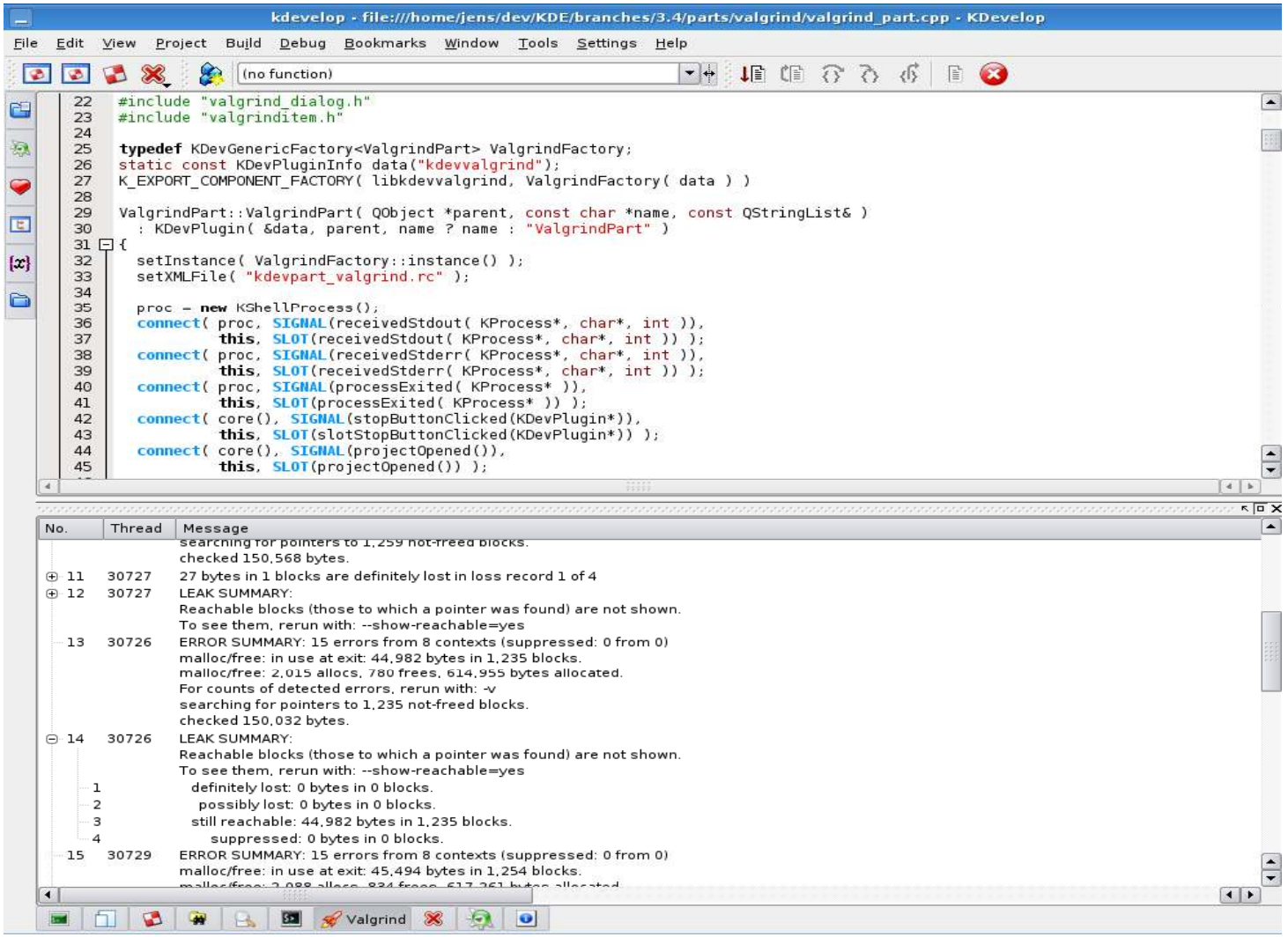
Valgrind is a suite of simulation-based debugging and profiling tools for programs running on Linux

The system consists of a core, which provides a synthetic CPU in software, and a series of tools, each of which performs some kind of debugging, profiling, or similar task

Various tools present in Valgrind are:

- Memcheck detects memory-management problems in programs
- Cachegrind is a cache profiler
- Helgrind finds data races in multithreaded programs
- Callgrind is a program profiler
- Massif is a heap profiler
- Lackey is a simple profiler and memory tracer

Valgrind: Screenshot



Insure++

Insure++ is a runtime memory analysis and error detection tool for C and C++ that automatically identifies a variety of difficult-to-track programming and memory-access errors, along with potential defects and inefficiencies in memory usage

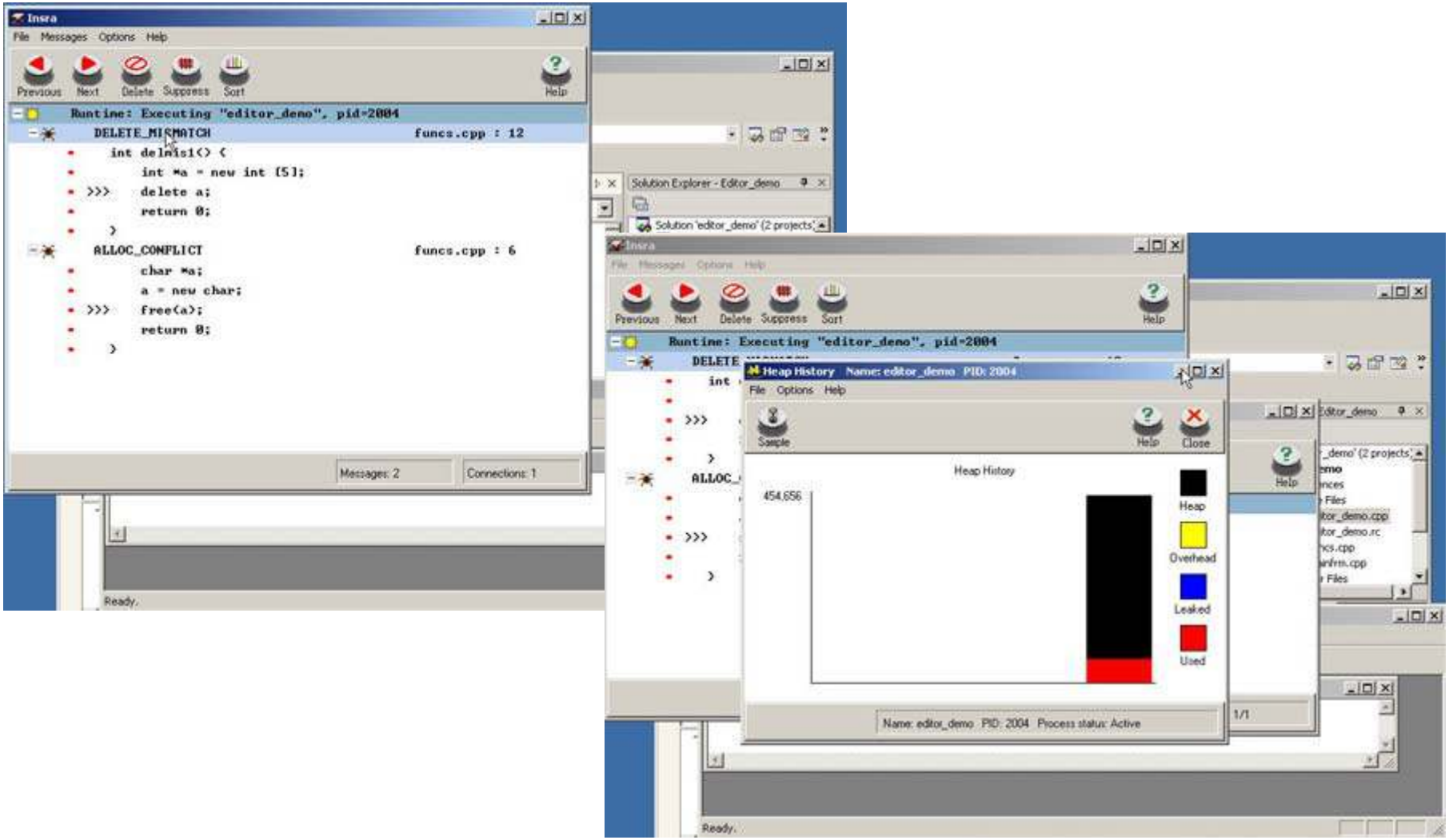
Errors that Insure++ detects include:

- Corrupted heap and stack memory
- Use of uninitialized variables and objects
- Array and string bounds errors on heap and stack
- Use of dangling, NULL, and uninitialized pointers
- All types of memory allocation and free errors or mismatches
- All types of memory leaks
- Type mismatches in global declarations, pointers,and function calls
- Some varieties of dead code (compile-time)

Insure++: Features

- Detection of memory corruption on heap and stack
- Detection of uninitialized variables, pointers, and objects
- Detection of memory leaks and other memory allocation/ free errors
- STL checking** for proper usage of STL containers, and related memory errors
- Compile-time checks for type- and size-related errors
- Runtime tracing of function calls
- GUI and command line interface
- Memory error checking in 3rd party static and dynamic libraries
- Direct interfaces with Visual Studio debugger

Insure++: Screenshot



Buffer Overflow Protection Solution: Libsafe

Libsafe is a library which re-writes some sensitive libc functions (strcpy, strcat, sprintf, vsprintf, getwd, gets, realpath, fscanf, scanf, sscanf) to prevent any overflow caused by a misuse of any one of them

It launches alerts when an overflow attempt is detected

Libsafe intercepts the calls to the unsafe functions and uses its own implementation of the function instead

While keeping the same semantic, it adds detection of the bound violations

Comparing Functions of libc and libsafe

Some functions of libc are unsafe because they do not check the bounds of a buffer

Implementation of strcpy by libc:

```
char * strcpy(char *
dest,const char *src)
{
char *tmp = dest;
while ((*dest++ = *src++) !=
'\0')
/* nothing */;
return tmp;
}
```

The size of the dest buffer is not a factor for deciding whether to copy more characters or not

Implementation of strcpy by libsafe:

```
char *strcpy(char *dest, const
char *src)
{
...
if ((len = strnlen(src,
max_size)) == max_size)
_libsafe_die("Overflow caused by
strcpy()");
real_memcpy(dest, src, len + 1);
return dest;
}
```

Function libsafe_die stops the process if strlen returns max_size (attempt to buffer overflow)

Simple Buffer Overflow in C

Vulnerable C Program overrun.c

```
#include <stdio.h>
main() {
  char *name;
  char *dangerous_system_command;
  name = (char *) malloc(10);
  dangerous_system_command = (char *) malloc(128);
  printf("Address of name is %d\n", name);
  printf("Address of command is %d\n", dangerous_system_command);
  sprintf(dangerous_system_command, "echo %s", "Hello world!");
  printf("What's your name?");
  gets(name);
  system(dangerous_system_command);
}
```

Code Analysis

The first thing the program does is declare two string variables and assign memory to them

The "**name**" variable is given 10 bytes of memory (which will allow it to hold a 10-character string)

The "**dangerous_system_command**" variable is given 128 bytes

You have to understand that, in C, the memory chunks given to these variables will be located directly next to each other in the virtual memory space given to the program

Code Analysis (cont'd)

- To compile the overrun.c program
- Run this command in Linux:

```
gcc overrun.c –o overrun
[XX]$ ./overrun
Address of name is 134518696
Address of command is 134518712
What's your name?xmen
Hello world!
[XX]$
```

- The address given to the **"dangerous_system_command"** variable is 16 bytes from the start of the "name" variable
- The extra 6 bytes are overhead used by the "malloc" system call to allow the memory to be returned to general usage when it is freed

Code Analysis (cont'd)

The "**gets**", which reads a string from the standard input to the specified memory location, does not have a "length" specification

This means it will read as many characters as it takes to get to the end of the line, even if it overruns the end of the memory allocated

Knowing this, a attacker can overrun the "**name**" memory into the "**dangerous_system_command**" memory, and run whatever command he/she wishes

Code Analysis (cont'd)

```
[XX]$ ./overrun
Address of name is 134518696
Address of command is 134518712
What's your name?01234567891234567cat /etc/passwd

root:x:0:0:root:/root:/bin/bash
bin:x:1:1:bin:/bin:
daemon:x:2:2:daemon:/sbin:
adm:x:3:4:adm:/var/adm:
lp:x:4:7:lp:/var/spool/lpd:
sync:x:5:0:sync:/sbin:/bin/sync
shutdown:x:6:0:shutdown:/sbin:/sbin/shutdown
halt:x:7:0:halt:/sbin:/sbin/halt
mail:x:8:12:mail:/var/spool/mail
```

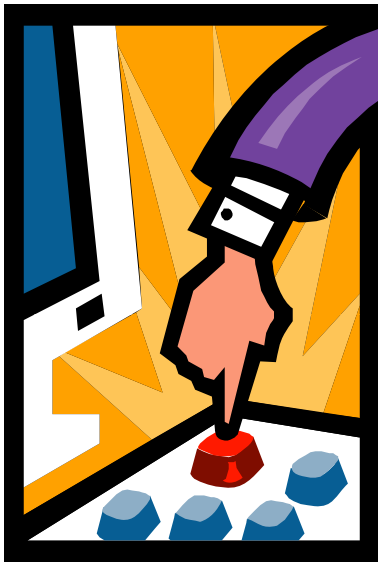
What Happened Next

Since the project was running behind schedule, he had to hurry through testing.

Tim had worked with the same team for his previous projects, and all of the other projects had successful conclusions. Therefore, he thought that nothing would possibly go wrong with this one. This notion made him overconfident about the testing of this project.

But this time, he was not lucky. The web server of the client company had succumbed to a buffer overflow attack. This was due to a flaw in coding because bounds were not checked.

Is Tim's decision justified?



Summary

- A buffer overflow occurs when a program or process tries to store more data in a buffer (temporary data storage area) than it was intended to hold
- Buffer overflow attacks depend on: the lack of boundary testing, and a machine that can execute a code that resides in the data/ stack segment
- Buffer overflow vulnerability can be detected by skilled auditing of the code as well as boundary testing
- Countermeasures include checking the code, disabling stack execution, safer C library support, and using safer compiler techniques
- Tools like stackguard, Immunix, and vulnerability scanners help in securing systems

"When I told him we need to increase our bandwidth, he hired six fat tuba players."