Reverse Engineering Linux ELF Binaries on the x86 Platform

The University of Adelaide

Reverse Engineering

- Reverse Engineering is the process of examining and probing a compiled program, and determining the original design of the program
- The documentation a reverse engineer writes can be used to
  - Document the purpose of an unknown program (the target)
  - Recreate source code for the target
  - Implement another program that is compatible with the target programs communication or data format
  - Add or disable features of the target
  - Discover and document undocumented behaviour of the target

The Honeynet Reverse Challenge

- The HoneyNet Project aims to discover and document the current methods and tools being used by crackers
  - To acheive this objective, the project puts monitored "Honeypot" machines on the Internet and waits until they are cracked
  - Once the honeypot is cracked, the attackers tools and methods are examined
    - Tools are recovered from the compromised honeypot and reverse engineered
    - Network dumps are examined to determine the crackers actions
    - Modified kernels and shells record the attackers keystrokes
    - The attackers methods are studied for new attack patterns
    - Future attack trends are predicted
- Scan of the Month
- The HoneyNet Projects books
- http://www.honeynet.org/

Methodology

- Determine your objective
- Identify relevant code or data
    - Dead Listing
    - Tracing Program Execution
    - Examining Network Traffic
- Document program design
- Repeat

Determine Your Objective

There are many approaches

- Your objective will determine your methodology
    - Quick focussed exploration
        - Determine key functions and data
        - Use deadlisting and debugging to find calls to key functions or modifications of key data
        - Examine and document interesting functions from previous step
    - In depth analysis
        - Generate call tree
        - Examine and document functions either top down or bottom up
        - Test documentation by running test data through the target
        - Rewrite target in high level language

Example of a quick focussed approach

- A Shareware Windows disassembler
- Shareware version popped up a message on start up, and stopped working after X days
  - Objective: Remove the time check and message
  - Method
    - Disassemble the disassembler
    - Find references to string stating that the program would expire
    - Examine disassembly near point where message box is displayed
    - Modify code at time check using a hex editor to jump past shareware tests
  - Result: unwanted feature removed

Example of an in depth analysis

- The Honeynet Reverse Challenge
  - Objective: Determine and document program behaviour
  - Method
    - Disassemble program
    - Find main()
    - Work in from main() labelling functions
    - Determine packet format and encoding by examining functions that handle the packets
    - Build test client and confirm server execution follows what has already been discovered
    - Use the test client to probe unknown functionality while tracing the server in a debugger
    - Document the-binary and network protocols
  - Result: Program and network protocol documented

Identify relevant code or data

Basic Information

- Basic Unix utilities such as 'file' and 'strings' reveal information about an unknown binary
  - file reveals that the-binary is a statically linked and stripped ELF binary, for the Intel x86 platform

```
 [slide@host]$ file the-binary
the-binary: ELF 32-bit LSB executable, Intel 80386, version 1,
            statically linked, stripped
```

  - strings reveals a few clues about the-binary's purpose and the platform it was built on

```
 [slide@host]$ strings the-binary
[mingetty]
/tmp/.hj237349
/bin/csh -f -c "%s" 1> %s 2>&1
TfOjG
...
/bin/sh
/bin/csh -f -c "%s"
...
@(#) The Linux C library 5.3.12
...
yplib.c,v 2.6 1994/05/27 14:34:43 swen Exp
```

Introduction to the ELF File Structure

Inside an ELF executable file

File Offset	File Section	Virtual Address
0x00000	ELF Header (readelf -h)	0x8048000
0x00024	Program Header Table (readelf -l)	0x8048024
0x00080	Text Section (contains segments: .init .text __libc_subinit .fini .rodata) Read Only, Executable 0x24222 bytes	0x8048080
0x24228	Data Section (contains segments: .data .ctors .dtors .bss) Read Write 0xc094 bytes	0x806d228

ELF Symbol Table

- Programmers usually make use of libraries of functions
- Program source code is compiled to create the program binary, which the operating system can then run
- To save space and memory, the library functions that programmers use are stored in a seperate file, referred to as a shared library
- When a program is about to be run, the operating system runs another program, called a dynamic linker, which loads the required libraries into memory and links the program to them
- The dynamic linker uses the ELF binaries symbol table to determine which libraries to load, and to modify the loaded program so that it knows how to access the library functions
- **The symbol table lists library functions that a program uses**

ELF Static Binaries

- Library functions can be built into the binary by the compiler, rather than having the dynamic linker load them
  - This enables the binary to run on computers that do not have the necessary shared libraries installed
  - The downside of this portability is that the library function cannot be upgraded by simply replacing the shared library. Instead, the binary has to be recompiled against the new library. This can be a security problem if a hole is found in the library function.
- A static binary no longer needs a symbol table, as it does not load functions from shared libraries. It may have one anyway, listing the functions that are included inside the binary. These function names help identify functions when you disassemble the binary.
- The binary can be stripped of its symbol table, along with debugging information such as function names, to reduce its size

Rebuilding a stripped symbol table

- The Reverse Challenge binary is a static binary, so it does not use any external libraries
- It has been stripped, so we do not know what the functions built into the binary are
- The symbol table can be rebuilt
  - 1. Generate a fingerprint of the first few bytes of each library function that you think the binary include
  - 2. Generate a list of functions within the binary
  - 3. Create a fingerprint of the first few bytes of each function
  - 4. Compare each function fingerprint from the binary with the library function fingerprint set, and update the binaries symbol table with the name of any matching functions
- The 'dress' utility, that comes with the Fenris debugging package, can automatically rebuild symbol tables. It comes with a collection of library function fingerprints.

Program and Data Structures

Loading an ELF binary

- When an ELF binary is loaded into memory, the operating system loads each section into a different area of virtual memory. It also allocates any memory required by the .bss (uninitialised data) section. Finally, it creates a stack for short term storage.

```
[root@host]# ps axc | fgrep the-binary
  987 ?        S      0:00 the-binary
[root@host]# cat /proc/987/maps
// The text (code) section is mapped at 0x08048000
08048000-0806d000 r-xp 00000000 16:06 598925
           /home/slide/src/rev-challenge/reverse/the-binary
// The data section is mapped at 0x0806d000
0806d000-0807a000 rw-p 00024000 16:06 598925
           /home/slide/src/rev-challenge/reverse/the-binary
// The uninitialised data segment .bss is allocated at 0x0807a00
0807a000-0807f000 rwxp 00000000 00:00 0
// The stack is allocated at 0xbfffa000
bfffa000-c0000000 rwxp ffffb000 00:00 0
```

The Stack

- The stack is a First in Last out buffer in memory that is used to store local variables and function call arguments
- Two CPU registers (variables) keep track of the stack:
  - ESP Stack Pointer: points to the bottom of the stack
  - EBP Branch Pointer: points to top of the stack for the current function
- Two assembly instructions are provided for manipulating the stack:
  - push <register>: Stores a value onto the bottom of the stack, and decrements ESP
  - pop <register>:   Retrieves a value from the bottom of the stack, and increments ESP
- The stack is often manipulated to directly using offsets from EBP, rather than using push and pop

Function Calls

- When a function is about to get called, its arguments are put onto the bottom of the stack.
- The assembly instruction *call <address>* is used to call the function. Call pushes the return address onto the bottom of the stack
- The function then pushes EBP onto the stack, points EBP to the bottom of the stack (EBP), and adjusts ESP to make space for local variables
  - Arguments to the function can now be referred to using positive offsets from EBP
  - Local variables can now be refered to using negative offsets from EBP
- Before returning from the function, ESP is set back to EBP and EBP is popped off the stack
- The assembly instruction *ret* (return) returns to the location pointed to by the value at the bottom of the stack

Function Calls

- Example: the-binary inside main()
  - __init() has called main(int argc, char *argv[], char *envp[]):

```
__entry_point__()
{
...
    main(int argc, char *argv[], char *envp);
...
}

main(int argc, char *argv[], char *envp)
{
...
}
```

Function Calls

- Example: the-binary inside main()
  - At this point, the stack looks like this:

```
ESP=0xbfffb604
EBP=0xbffffb04
Stack:
// Bottom of stack is at 0xbfffb604
0xbffffb04:     0xbffffb18 // EBP value for return to __init
0xbffffb08:     0x080480eb // Return address from main()
0xbffffb0c:     0x00000001 // argc = 1
0xbffffb10:     0xbffffb24 // argv[] = ("the-binary", "")
0xbffffb14:     0xbffffb2c // envp[] = ("PWD", "/home/slide/..."
// End of arguments to main()
0xbffffb18:     0x00000000 // EBP value for return to __init
0xbffffb1c:     0x00000000 // Return address from __init()
// Top of stack is at 0xc0000000
```

  - Note that __init() does not return, so no return value is stored on the stack

Conditionals and Loops

- Compilers generate "signature" code for different program structures, such as conditional statements (if, else, switch) and loops (for, do while)
- By recognising this assembly code, we can guess what the original program structure looked like
  - The assembly representation of various C instructions changes, depending on the compiler and the level of optimisation it does
- With concentration and time, we can reconstruct the original program source
- A good paper on this is StrIkeR_MaN's tutorial "Introduction to Reverse Engineering Software in Linux"
  - http://www.acm.uiuc.edu/sigmil/RevEng/t1.htm

Automating Program Structure Analysis

- Reverse engineering the structure of a function is slow and difficult
- Why not automate it?
  - (because optimisation makes that difficult)

Automating Program Structure Analysis

- Reverse engineering the structure of a function is slow and difficult
- Why not automate it?
  - (because optimisation makes that difficult)
- REC - The Reverse Engineers Compiler
  - Disassembles and decompiles executable to C like source
  - Works wonders on the Reverse Challenge binary
  - http://www.backerstreet.com/rec/rec.htm

Examining Deadlisting

Finding Relevant Functions

- Remove known functions to remove clutter
- Examine the call tree
- Search for calls to key functions
- Search for accesses to key data

The Call Tree - an in depth analysis tool

- A call tree shows which functions are called from within each function
- Drawing a call familiarises you with the program structure, and provides a quick reference as to the likely behaviour of each function

```
__entry_point__()
  _exit()
  main()
    geteuid()
    fork()
    socket()
    receive()
    decrypt()
    cmd_01__status()
      encrypt()
      rand()
      send_response()
    cmd_02__configure()
...
```

Calls to Key Functions - a focussed examination method

- By searching for calls key functions, we can quickly identify interesting functions that are worth more investigation
- The key functions you are interested in will depend on the functionality you are investigating
- For example, if you are interested in the format of packets that the-binary accepts, you would start by searching for calls to the recv() function
- Once you find the call to recv() in main(), you quickly find decode() is the next function call!

Accesses to Key Data - a focussed examination method

- By searching for manipulation of key data, we can quickly identify interesting functions that are worth more investigation
- A disassembler that cross references data with variables in code is really handy for this
- For example, the code to send responses to the controller of the-binary uses an array of IP addresses as a list of addresses to send response packets to
- By searching for accesses to this array, we quickly discover that command 2 configures this list of response addresses

Limitations of Dead Listing

- Cannot see inside encrypted/encoded code or data
- May miss code hidden in sections other than .text
- Cannot easily examine variable values specified points of execution
- May fall foul of anti-disassembly tricks

Tracing Program Execution

About Execution Tracing

- Provides opportunity to probe data values or program behaviour
- When dealing with unknown binaries, this is a stupid, but necessary, method
  - Hit code hidden in library functions
  - Lose control of execution
  - Hit anti-debugger measures
  - Accidentally launch attacks or modify system
- Use a virtual machine
  - Easier to monitor
  - Filesystem easy to restore
  - Examples include User Mode Linux (linux under linux) or VMWare

Before Execution of an Untrusted Binary

- Baseline filesystem using tripwire or similar
- Take a snapshot of network state (netstat, nmap)
- Setup monitoring of network activity on a seperate machine
- Harden the monitoring hosts
- Disconnect from live networks

Gathering Information about a Running Program

- /proc
  - Process Status (status)
  - Command Line (cmdline)
  - Environment (environ)
  - Memory Map (maps)
  - Open File Descriptors (fd)
- Tracing
  - System Calls (strace)
  - Library Calls (ltrace)
  - Debugging (gdb, aegir, pice)
- Network Profile
  - Network Footprint (netstat, nmap, lsof)
  - Network Activity (ethereal, tcpdump)

Tracing using Strace

- strace shows system calls

```
[slide@host]$ strace -fxi ./the-binary
[????????] execve("./the-binary", ["./the-binary"], [/* 21 vars
[080480b6] personality(PER_LINUX)       = 0
[08057216] geteuid()                    = 500
[08057562] _exit(-1)                    = ?
[slide@host]$
```

Tracing using Strace

- strace shows system calls

```
[root@host]# strace -fxi ./the-binary
[????????] execve("./the-binary", ["./the-binary"], [/* 24 vars
[080480b6] personality(PER_LINUX)       = 0
[08057216] geteuid()                    = 0
[080574f9] sigaction(SIGCHLD, {SIG_IGN}, {SIG_DFL}, 0x40086558)
[080571f2] fork()                       = 971
[pid   970] [08057562] _exit(0)         = ?
[08057346] setsid()                     = 971
[080574f9] sigaction(SIGCHLD, {SIG_IGN}, {SIG_IGN}, 0x80575a8) =
[080571f2] fork()                       = 972
[pid   972] [08057142] chdir("/")       = 0
[pid   972] [0805716e] close(0)         = 0
[pid   972] [0805716e] close(1)         = 0
[pid   972] [0805716e] close(2)         = 0
[pid   972] [08057452] time(NULL)       = 1029748047
[pid   972] [08056d1e] socket(PF_INET, SOCK_RAW, 0xb /* IPPROTO_
[pid   972] [080574f9] sigaction(SIGHUP, {SIG_IGN}, {SIG_DFL}, 0
[pid   972] [080574f9] sigaction(SIGTERM, {SIG_IGN}, {SIG_DFL},
[pid   972] [080574f9] sigaction(SIGCHLD, {SIG_IGN}, {SIG_IGN},
[pid   972] [080574f9] sigaction(SIGCHLD, {SIG_IGN}, {SIG_IGN},
[pid   972] [08056b74] recv(0,
```

Examining the State of a Process

- About the raw socket
  - The socket is listening for packets that are using IP protocol 11
  - This is a transport layer protocol
  - Other transport layer protocols include TCP and UDP
  - Transport layer protocols sit on top of network layer protocols such as IP (in this case) or IPX
- Protocol 11 is reserved for Network Voice Protocol, a protocol that is not widely used and is probably dead
- Packets using protocol 11 will bypass certain firewalls, for example the RedHat 7.2 firewall blocks most TCP and UDP, however protocol 11 is allowed through by default

Examining the State of a Process

- About the raw socket
  - The socket is listening for packets that are using IP protocol 11
  - This is a transport layer protocol
  - Other transport layer protocols include TCP and UDP
  - Transport layer protocols sit on top of network layer protocols such as IP (in this case) or IPX
- Protocol 11 is reserved for Network Voice Protocol, a protocol that is not widely used and is probably dead
- Packets using protocol 11 will bypass certain firewalls, for example the RedHat 7.2 firewall blocks most TCP and UDP, however protocol 11 is allowed through by default
- **We now have enough information to block the-binary's control channel at our firewall**

Examining the State of a Process

- Discovering open files

```
[root@host]# /usr/sbin/lsof -p 972
COMMAND   PID USER   FD   TYPE DEVICE   SIZE   NODE NAME
the-binar 972 root  cwd    DIR   22,6   4096      2 /
the-binar 972 root  rtd    DIR   22,6   4096      2 /
the-binar 972 root  txt    REG   22,6 205108 598925
                   /home/slide/src/rev-challenge/reverse/the-bi
the-binar 972 root    0u   raw                   5345
                   00000000:000B->00000000:0000 st=07
```

- The raw entry is the raw socket, listening on protocol 11 (0x0B)
- Why has it opened the-binary?

Examining the State of a Process

- Discovering open files

```
[root@host]# /usr/sbin/lsof -p 972
COMMAND   PID USER   FD   TYPE DEVICE   SIZE   NODE NAME
the-binar 972 root  cwd    DIR   22,6   4096      2 /
the-binar 972 root  rtd    DIR   22,6   4096      2 /
the-binar 972 root  txt    REG   22,6 205108 598925
                   /home/slide/src/rev-challenge/reverse/the-bin
the-binar 972 root    0u   raw                 5345
                   00000000:000B->00000000:0000 st=07
```

- The raw entry is the raw socket, listening on protocol 11 (0x0B)

- Why has it opened the-binary?

```
[root@host]# cat /proc/972/maps
08048000-0806d000 r-xp 00000000 16:06 598925
                  /home/slide/src/rev-challenge/reverse/the-bina
0806d000-0807a000 rw-p 00024000 16:06 598925
                  /home/slide/src/rev-challenge/reverse/the-bina
0807a000-0807f000 rwxp 00000000 00:00 0
bfffb000-c0000000 rwxp ffffc000 00:00 0
```

- Every ELF program has itself open, as its .code and .data sections are mapped into memory with mmap()

Examining the State of a Process

- Discovering network sockets
  - (Assuming netstat has not been replaced)
- Take a netstat baseline

```
[root@host]# netstat -ln --protocol=inet
Active Internet connections (only servers)
Proto Recv-Q Send-Q Local Address           Foreign Address
tcp        0      0 0.0.0.0:6000            0.0.0.0:*
tcp        0      0 0.0.0.0:22              0.0.0.0:*
```

Examining the State of a Process

- Discovering network sockets
  - (Assuming netstat has not been replaced)
- Take a netstat baseline

```
[root@host]# netstat -ln --protocol=inet
Active Internet connections (only servers)
Proto Recv-Q Send-Q Local Address           Foreign Address
tcp        0      0 0.0.0.0:6000            0.0.0.0:*
tcp        0      0 0.0.0.0:22              0.0.0.0:*
```

- Run the program

```
[root@host]# ./the-binary
```

Examining the State of a Process

- Discovering network sockets
  - (Assuming netstat has not been replaced)
- Take a netstat baseline

```
[root@host]# netstat -ln --protocol=inet
Active Internet connections (only servers)
Proto Recv-Q Send-Q Local Address           Foreign Address
tcp        0      0 0.0.0.0:6000            0.0.0.0:*
tcp        0      0 0.0.0.0:22              0.0.0.0:*
```

- Run the program

```
[root@host]# ./the-binary
```

- Compare netstat output

```
[root@host]# netstat -ln --protocol=inet
Active Internet connections (only servers)
Proto Recv-Q Send-Q Local Address           Foreign Address
tcp        0      0 0.0.0.0:6000            0.0.0.0:*
tcp        0      0 0.0.0.0:22              0.0.0.0:*
raw        0      0 0.0.0.0:11              0.0.0.0:*
```

- This raw socket matches what strace and lsof show

Using a Debugger

- Tools available in debugging enviroments
  - Breakpoints to pause execution
    - When execution reaches a specified point
    - When specified memory is accessed of modified
  - Examine memory and CPU registers
  - Modify memory and execution path
- Tools available in advanced debugging enviroments
  - Attach comments to code or data
  - Track higher level logic
    - Level of function call nesting
    - Memory, map and file descriptor tracking
  - Function fingerprinting and naming
  - Data structure templates and naming

Using a Debugger - Quick focussed exploration

- Example: Examining the password handling of the-binary's bindshell
  - Previous examination had revealed that a particular packet sent to the-binary caused it to spawn a shell that listened on port 23281/TCP, and that it required password
  - A quick examination of the relevant function in the REC disassembly revealed that the string "TfOjG" was somehow compared to the password the user enters
  - Telnetting to port 23281 and entering TfOjG did not give access to the bindshell
  - While waiting for the user to enter a password, the bindshell function would be blocked in a recv() call
  - I decided to use a debugger to determine what was happening to the password I was entering
  - I knew that the-binary and its children had a process name of [mingetty]

Using a Debugger - Quick focussed exploration

- Example: Examining the password handling of the-binary's bindshell
    - Modified REC disassembly of the bindshell function of the-binary (at 0x08048984)

```
client = accept( sockfd, raddrptr, raddrlenptr);
if(client != 0) {
    if(fork() != 0) { goto L08048984; }
    recv(client, buffer, bufferlen, 0);
    ebx = 0;
    do {
        al = buffer[ebx];
        if(al == 0xa || al == 0xd) {
            buffer[ebx] = 0;
        } else {
            buffer[ebx]++;
        }
    } while(++ebx);
    if(memcmp(buffer, "TfOjG", 6) > 0) {
        send(client, escapecode, 4, 0);
        close(client);
        exit(1);
    }
```

Using a Debugger - Quick focussed exploration

- Example: Examining the password handling of the-binary's bindshell
  - Request that the-binary launches a bindshell

```
[root@host]# ./the-client -i tap0 -s 192.168.32.1
               -d 192.168.32.32 bindshell
```

Using a Debugger - Quick focussed exploration

- Example: Examining the password handling of the-binary's bindshell
  - Request that the-binary launches a bindshell

```
[root@host]# ./the-client -i tap0 -s 192.168.32.1
              -d 192.168.32.32 bindshell
```

  - Telnet to the bindshell to reach recv()

```
[root@host]# telnet 192.168.32.32 23281
```

Using a Debugger - Quick focussed exploration

- Example: Examining the password handling of the-binary's bindshell
  - ◆ Request that the-binary launches a bindshell

```
[root@host]# ./the-client -i tap0 -s 192.168.32.1
              -d 192.168.32.32 bindshell
```

  - ◆ Telnet to the bindshell to reach recv()

```
[root@host]# telnet 192.168.32.32 23281
```

  - ◆ Attach a debugger

```
target# ps ax | fgrep mingetty
  454 ?        S      0:00 [mingetty]
  507 ?        S      0:00 [mingetty]
  508 ?        S      0:00 [mingetty]
target# gdb
(gdb) attach 508
Attaching to process 508
0x08056b74 in ?? ()
(gdb) bt                    // Examine stack
#0  0x08056b74 in ?? () // recv()
#1  0x080489cf in ?? () // bindshell function
#2  0x080480eb in ?? () // main()
```

Using a Debugger - Quick focussed exploration

- Example: Examining the password handling of the-binary's bindshell
  - Set breakpoint on password compare (memcmp)

```
(gdb) disassemble 0x080489cf 0x08048a1b
Dump of assembler code from 0x80489cf to 0x8048a1b:
0x80489cf:      xor    %ebx,%ebx
0x80489d1:      add    $0x10,%esp
0x80489d4:      mov    0xffffbc44(%ebx,%ebp,1),%al
...
0x8048a0f:      mov    $0x6,%ecx
0x8048a14:      cld
0x8048a15:      test   $0x0,%al
0x8048a17:      repz cmpsb %es:(%edi) ("TfOjG"),%ds:(%esi) (buff
0x8048a19:      je     0x8048a44
End of assembler dump.
(gdb) break *0x8048a0f
Breakpoint 1 at 0x8048a0f
(gdb) cont
Continuing.
```

Using a Debugger - Quick focussed exploration

- Example: Examining the password handling of the-binary's bindshell
  - Type the password "TfOjG" into the telnet session, the breakpoint will be reached

```
(gdb) cont
Continuing.

Breakpoint 1, 0x08048a0f in ?? ()
(gdb) x/6c $edi  // Examine the password
0x8067617:      84 'T'  102 'f' 79 'O'  106 'j' 71 'G'  0 '\000
(gdb) x/6c $esi  // And the encoded buffer
0xbfffb748:     85 'U'  103 'g' 80 'P'  107 'k' 72 'H'  0 '\000
```

Using a Debugger - Quick focussed exploration

- Example: Examining the password handling of the-binary's bindshell
  - Type the password "TfOjG" into the telnet session, the breakpoint will be reached

```
(gdb) cont
Continuing.

Breakpoint 1, 0x08048a0f in ?? ()
(gdb) x/6c $edi  // Examine the password
0x8067617:      84 'T'  102 'f' 79 'O'  106 'j' 71 'G'  0 '\000
(gdb) x/6c $esi  // And the encoded buffer
0xbfffb748:     85 'U'  103 'g' 80 'P'  107 'k' 72 'H'  0 '\000
```

  - The password we entered, "TfOjG", has been turned into "UgPkH"
  - Re-examining the REC disassembly reveals that the entered password is rotated one character, so TfOjG becomes UgPkH
  - Therefore, the correct password is SeNif

Using a Debugger - Quick focussed exploration

- Example: Examining the password handling of the-binary's bindshell

```
[root@host]# telnet 192.168.32.32 23281
Trying 192.168.32.32...
Connected to 0.
Escape character is '^]'.
SeNiF
echo hi
hi
^]
telnet> close
Connection closed.
```

Examining Network Traffic

Capturing Network Traffic

- Network traffic can be captured and examined using a sniffer
  - Sniffers include tcpdump, ethereal and snort
- In a switched environment, arp tools can help you capture packets that are otherwise not sent to you
  - Example arp tools include DugSongs arpspoof, or arp-sk
  - It is easier to use a hub though...

Manipulating Network Traffic

- A proxy may help you manipulate traffic between a client and a server
  - Start with an existing proxy, such as udpproxy, and modify it to intercept the packets you wish to modify
- The following iptables rule, from http://www.thoughcrime.org/ie.html, will let you transparently proxy certain connections, provided your proxy is listening on the specified port:
  - iptables -t nat -A PREROUTING -p tcp --source-port 1024:5000 --destination-port 443 -j REDIRECT --to-ports <$listenPort>

Prototyping Network Clients and Servers

- Using existing network libraries may speed development up
  - libpcap: captures packets
  - libnet: generates packets
- Modules to interface with these libraries from high level languages such as Perl are probably available

Document program design

Take notes as you go

- Main features of examined code or data
  - Functionality
  - Algorithms
  - Relationship to other programs
  - Bugs
  - Data structures
  - Packet structures
- For examples, visit http://project.honeynet.org

Questions/Comments?

Links

Links - Papers

```
Honeynet
--------
Honeynet Project:
  http://www.honeynet.org/

Reference
---------
ELF Specification
  Text version with error corrections
    http://www.muppetlabs.com/~breadbox/software/ELF.txt
    http://www.muppetlabs.com/~breadbox/software/
  PDF version
    http://developer.intel.com/vtune/tis.htm
x86 Instruction reference
  Intel
    http://www.intel.com/design/pro/manuals/243191.htm
Linux syscall reference
  http://world.std.com/~slanning/asm/syscall_offline.html
```

Links - Papers

```
Tutorials
---------
Tutorials from Fravia
  http://www.woodmann.com/fravia/student.htm
  http://tsehp.cjb.net/
Gij's IDA tutorial
  http://home.online.no/~reopsahl/files/gij!ida.txt
Tutorials from LinuxAssembly.org
  Startup state of Linux/i386 ELF binary
    http://linuxassembly.org/startup.html
  Self modifying code under Linux
    http://linuxassembly.org/self.html
Introduction to Reverse Engineering software in Linux: Striker N
  http://www.acm.uiuc.edu/sigmil/RevEng/t1.htm
Linux Assembly howto
  http://www.tldp.org/HOWTO/Assembly-HOWTO/
```

Links - Papers

```
Articles
--------
Phrack
  http://www.phrack.com/archives/
Linux Viruses, ELF File Format
  http://download.nai.com/products/media/vil/pdf/mvanvoers_VB_co
Cheating the ELF: Subversive Dynamic Linking to Libraries the gr
  http://downloads.securityfocus.com/library/subversiveld.pdf
Papers by Silvio Cesare
  Kernel Function Hijacking
    http://www.big.net.au/~silvio/kernel-hijack.txt
  Linux Anti Debugging tricks - Fooling the debugger
    http://www.big.net.au/~silvio/linux-anti-debugging.txt
  etc...
```

Links - Tools

```
Disassembly
-----------
Fenris lcamtuf@bos.bindview.com
  http://razor.bindview.com/tools/fenris/index.html
REC
  http://www.backerstreet.com/rec/rec.htm
BIEW
  http://sourceforge.net/project/showfiles.php?group_id=1475

In Kernel Debuggers
-------------------
PICE KlausPG@SonicBLUE.com
  http://pice.sourceforge.net/downloads.html
The-Dude
  http://sourceforge.net/projects/the-dude

Development
-----------
NASM
  http://nasm.sourceforge.net/
LibNet
  http://www.packetfactory.net/libnet/
LibPCap
  http://www.tcpdump.org/
```

Links - Tools

```
Tracing
-------
User Mode Linux
  http://user-mode-linux.sourceforge.net/
VMWare
  http://www.vmware.com/download/workstation.html
  Get a 30 day license at
    http://www.vmware.com/vmwarestore/newstore/wkst_eval_login.j

ltrace
  http://packages.debian.org/unstable/utils/ltrace.html

tripwire
  http://sourceforge.net/projects/tripwire/

The coroners toolkit
  http://www.porcupine.org/forensics/tct.html
  docs from http://www.rootprompt.org/article.php3?article=738

IDA Pro demo
  Get it from http://www.datarescue.com/idabase/

Data structure analysis
-----------------------
Stan
  http://www.roqe.org/stan/
```

Links - Tools

```
Network
-------
Ethereal
  http://www.ethereal.com/

Snort
  http://www.snort.org/
  source and docs

UDP Proxy
  http://sourceforge.net/projects/udpproxy/

Netcat
  http://www.atstake.com/research/tools/index.html#network_util

NMap
  http://www.insecure.org/nmap/
  link to http://www.linuxgazette.com/issue56/flechtner.html
```

Links - Tools

```
Debuggers IDE
-------------
Bastard Disassembly Environment (plus libdasm)
  http://bastard.sourceforge.net/

Anti Debugging
--------------
Burneye
  http://teso.scene.at/releases.php
  + lcamtufs analysis from
    http://216.239.33.100/search?q=cache:DovUnJaje3gC:lcamtuf.co
```