UNIVERSITY OF CAMBRIDGE

Department of Computer Science and Technology

Cybersecurity

Professor Frank Stajano

Professor of Security and Privacy, University of Cambridge

Founder and CEO, Cambridge Cyber Ltd

Computer Science Tripos — Part IB
Academic year 2025–2026
Easter term 2026

Stay legal. Don't break anything.

In this course I am going to teach you some attacks and I will encourage you to become proficient at them. These attacks can cause serious damage: cyber weapons aimed at critical infrastructure use many of these same techniques. *Do not launch these attacks on any system other than your own experimental setup.* Understand how to confine your experiments and do so. Ensure that no other system is damaged or affected by your experiments. Stay legal at all times. You will be liable to be sued or prosecuted otherwise, and claiming that you did it for coursework will never be accepted as a mitigating excuse.

Videos: http://frankstajanoexplains.com
Course web page: http://www.cl.cam.ac.uk/teaching/current/CySecurity/

Email about this course, from a `cam.ac.uk` address,
to the following special address, will be treated with higher priority:
`frank.stajano--cysec2026@cl.cam.ac.uk`

2026 edition

Revision 24 of 2026-05-03 06:22:37 +0100 (Sun, 03 May 2026).

Contents

Chapter 0

Preliminaries

This course is a hands-on introduction to security for second-year computer science undergraduates at the University of Cambridge.

It is not intended just for future security specialists: it consists of foundational material that should be an essential part of the training of every computer professional. Many spectacular security incidents were (and still are) made possible by simple software vulnerabilities introduced by developers without this basic training. If all computer professionals learned to view the systems they develop through the lens of adversarial thinking, they would have a better understanding of how malicious actors can subvert and exploit them. Developers with such awareness would instinctively exercise more caution as they design and implement their systems. The digital society would be a safer place.

This is why all of you are studying this course, regardless of what branch of computer science you will later specialise in.

0.1 Course textbook

This course relies on the following excellent textbook:

> [CS3] Wenliang Du. *Computer Security: A Hands-on Approach. Third edition* 2022. ISBN 978-17330039-5-7.

When I took over this course I surveyed available textbooks and this[1] was the one that best aligned with my vision and goals. It is very much hands-on, as its subtitle promises, and it is supported by a wealth of well-designed and pedagogically valid online resources—the SEED labs (SEcurity EDucation). The book is self-published and could do with

[1] In its second edition back then.

a pass of professional copy-editing but it is nonetheless an outstanding resource. I recommend both the book and the SEED labs very highly.

Curiously, this textbook is offered in three variations:

- *Computer Security*,
- *Internet Security* and
- *Computer & Internet Security*,

with overlapping chapters[2]. All the topics in our syllabus are covered by the *Computer Security* variant, and chapter references in this handout refer to that book; but I also recommend the *Internet Security* book if you are interested in actually learning cybersecurity (and perhaps do graduate studies with me later on, and/or occasionally work for my company) rather than just passing the exam.

While the practical material in the textbook and in the SEED labs is precisely what I sought as the foundation for this course, I also wanted to give you some awareness of the bigger picture of security. I felt the book did not give you enough of that, so I am supplementing it with additional material on such topics as user authentication, human factors, physical security and risk management. But note that **this handout is not a summary of, nor a substitute for, the official course textbook.** You are highly advised to study on both.

0.2 Hands-on exercises (SEED labs)

The SEED labs[3] at https://seedsecuritylabs.org offer you a valuable chance to recreate practical attacks. These exercises will not be marked or assessed so there is no point in cheating, which includes looking for solutions on the web or getting an LLM to do the work. Do the exercises purely for your own education, without looking at other people's solutions, otherwise don't bother doing them at all, because in any case you'll get no course credit for them. Cybersecurity is a topic you only really learn through *doing*, by thinking hard about solving the problems yourself, not merely by reading/listening/watching/copying/replicating. I highly recommend you do the SEED labs *as you go along* in the course, as an aid to understanding rather than as a revision tactic.

[2] See the summary table at https://www.handsonsecurity.net/files/chapters/edition3/coverages.pdf. If you want to get *all* the chapters, buy the first two books (as I did), rather than the third. Up to the second edition there was a single book with all the content, but not any more (too many pages to bind).

[3] Here "lab" means "exercise" or "experiment".

The SEED labs are based on a preconfigured virtual machine, downloadable from the website at the address above, that exposes vulnerable services for you to attack in a controlled fashion, as well as all the tools required to implement the attack. Each lab then provides a `Labsetup.zip` file with further relevant material, sometimes also with a Docker image to simulate a network of several computers.

The SEED labs were originally designed for x86 processors. For all the practical examples in this course we shall stick to x86 only, since our focus is on concepts and there is no reason nor time to cover the detail of different sets of assembly languages, calling conventions and so forth. Until 2020, life was easier: Windows, Mac and Linux all ran on x86. But nowadays the Apple fans have laptops with Apple Silicon processors that cannot natively run an x86 virtual machine (VM). If you are in that situation you may either get yourself an old x86 computer that someone is discarding and install Linux on it and the SEED VM inside that (as I do); or you may instead use a SEED VM in the cloud[4].

We offer weekly practical helpdesk sessions to help you with the SEED labs. If, after your best efforts, you get really stuck anywhere (including, initially, setting up the VM on your computer or in the cloud), turn up at the designated times and a helpful and competent Teaching Assistant will help you sort things out. Note the dates, times and venue in your diary and make good use of this resource, in your own interest. You will not be graded nor awarded any ticks for either doing the SEED labs or attending helpdesk sessions.

0.3 Lectures

As per the Statutes and Ordinances of the University of Cambridge, I own the copyright and performance rights to my lectures. I recorded, edited, polished and published this whole course on my YouTube channel, `http://frankstajanoexplains.com`. The videos may now be enjoyed at no charge by any interested person in the world. Make the most of them. You may also ask questions there, pseudonymously if you prefer: unlike some colleagues who also publish their lectures but disable comments on them, I do my best to reply publicly to all genuine and pertinent questions. If there is anything you can't understand, please ask—maybe I was unclear, or sometimes even wrong. And when you like a video, please leave a thumbs up.

[4]Details at `https://seedsecuritylabs.org/labsetup.html`. Further details of local interest on the materials tab of the course web page at `https://www.cl.cam.ac.uk/teaching/current/CySecurity/materials.html`.

Please also fill out the feedback form when the course finishes, saying what you liked, not just what you disliked. Although it is impossible to please everyone all the time, and some requests conflict with pedagogical goals, I do listen to your feedback with care and have taken action on many useful suggestions from your predecessors. Constructive feedback is greatly appreciated and it has been helpful and encouraging in the past.

0.4 Syllabus and past exam questions

The course syllabus is found on the official course web page at https://www.cl.cam.ac.uk/teaching/current/CySecurity/ and is broadly reflected in the Table of Contents (page 5) of this handout. There is a copious supply of past exam questions at http://www.cl.cam.ac.uk/teaching/exams/pastpapers/ under Cybersecurity, Security, Introduction to Security, Security I and Security II but, before attempting an exam question, please check, perhaps with the help of your supervisor, that its content is still covered in this year's syllabus.

Going forward, the best way to prepare for the forthcoming exam is to gain practical experience by completing the SEED labs on your own, without using an LLM, without looking up the solutions anywhere. Those who do so will be greatly advantaged, both at the exam and, more importantly, in their professional career, whether they specifically pursue security or not.

0.5 Acknowledgements and history

I wrote the first version of this handout from scratch in 2021–2022 and then updated it incrementally in subsequent years. The Part 1b Security course was previously lectured by my esteemed colleague Dr Markus Kuhn, whose handout consisted not of prose but of slides with bullet points—a lecturing style that does not suit me very well. I am grateful for the inspiration provided by his material but I have not directly incorporated any of it in mine. You are still welcome to consult Dr Kuhn's handout as an additional resource from his web page while he still makes it available. I particularly recommend his collection of thought-provoking exercises at https://www.cl.cam.ac.uk/teaching/2021/Security/security-exercises.pdf, of which at least the following relate to topics in this year's syllabus: 1, 2, 3, 4, 5, 6, 7, 8, 14, 15, 16, 18, 19, 20, 22, 23.

0.6 Errata

Although I don't know where they are, my longtime experience as author makes me confident that this document still contains bugs. Consult the course web page for any updates or corrections. I am grateful to Jacky Kung, Elizabeth Ho, Jack Hughes and Arya Golkari for sending me useful comments, questions and bug reports. Whether you are a student or a supervisor, if you find anything that needs fixing then please email me at the address on the front page and I'll credit you in any future revisions (unless you tell me that you prefer anonymity). The responsibility for any remaining mistakes is still mine.

Chapter 1

Introduction

1.1 Challenging but fascinating

Cybersecurity is increasingly recognised as vital for commercial survival, national security, online business and for the preservation of appropriate privacy for individuals, including confidentiality of personal information. Protection against both criminal and state-sponsored attacks will need a large cohort of skilled individuals with an understanding of the principles of security and with practical experience of the application of these principles[1].

But this Part 1b Security course is not aimed merely at people who will specialise in security. As I mentioned in the opening preliminaries, many (most?) disastrous computer security incidents are the result of software developers lacking the adversarial mindset and the awareness that, once their system is deployed, there will be malicious actors who will try to poke holes in it for nefarious purposes. I thus believe that some basic security education should be part of the curriculum of every

[1] To help raise a new generation of cyber-defenders, I co-founded an international cybersecurity competition, the C2C (Cambridge to Cambridge) CTF with colleagues from MIT, and a national one, the Inter-ACE CTF between the UK's Academic Centres of Excellence in Cyber Security Research. See

> Frank Stajano, Graham Rymer, Michelle Houghton. "Raising a new generation of cyber-defenders". University of Cambridge Computer Laboratory Technical Report 922, June 2018.
>
> https://www.cl.cam.ac.uk/techreports/UCAM-CL-TR-922.pdf

I currently serve on the steering board of the subsequent edition of the C2C CTF, which has now been renamed "Country to Country" and has already been hosted by universities in UK, Israel, USA, Japan, Australia and Indonesia. I encourage you to participate in these and other CTFs to gain experience, have fun, learn from your peers and even, if you're particularly good, take home some medals and prize money. Watch this playlist on YouTube.

computer scientist. This is what I aim to do here.

This course will be challenging but I hope you will also find it fascinating. Security is about outsmarting intelligent adversaries who could strike anywhere. Any part of your system could become a target. Your devious adversaries will probe your system for weaknesses and look for the part that *they* find easiest to attack—which might not be the part you know or understand best. So one reason why this course may be more challenging than others in the Computer Science Tripos is because it requires you to have (or quickly acquire) some background knowledge in many seemingly unrelated topics. Computing is a very complex field, and no single human being, even a Turing Award winner, is able to understand absolutely everything about how a computer works at all levels, from the protocol messages exchanged between the web browser and the web server to the structure of the application program, the operating system calls, the innards of device drivers, the machine code executed by the processor, the circuitry making up the physical hardware, the device physics of the transistors and so on. As computer scientists, we use abstraction to dominate this complexity: when we focus on one layer of the technology stack, we take a simplified view of everything that happens underneath it. And in general that is a very good approach. But, in security, we cannot always afford this luxury and convenience.

1.2 Breaking abstraction

Here is an example from later on in the course. Imagine a program that sends you a present of either 5 or 10 pounds depending on the flip of a coin. In C, it might look something like Listing 1.1 on page 15.

You notice in lines 6–8 that it uses three variables: `name`, a 20-character string to hold your name; `reward`, an integer to hold the number of pounds to be paid out to you; and `payee`, an 18-character string to hold your sort code and account number. If you run the program and feed it some input, the interaction might look similar to this:

```
Your name? Honest Joe
Your sort code and bank account? 12-12-12 34563456
Congratulations, Honest Joe! I'll send 10 GBP to 12-12-12 34563456.
```

You will notice that the program only lets you write to the variables `name` (with the `gets(name)` function call in line 18) and `payee` (in line 20), but not to `reward`. And yet a devious attacker might find ways to increase the reward just by crafting suitable inputs for `name` and `payee`. Can you see how? If you think you can, why don't you stop reading, try

```c
#include <stdio.h> /* printf(), gets() */
#include <stdlib.h> /* rand(), srand() */
#include <time.h> /* time() */

int main() {
  char name[20];
  int reward;
  char payee[18];

  srand(time(NULL)); /* seed the RNG */

  if (rand() % 2) { /* flip a coin (odd or even) to pick reward */
    reward = 5;
  } else {
    reward = 10;
  }
  printf("Your name? ");
  gets(name);
  printf("Your sort code and bank account? ");
  gets(payee);

  printf("Congratulations, %s! I'll send %d GBP to %s.\n",
     name, reward, payee);
  return 0;
}
```

Listing 1.1: buffer–overflow–0.c

compiling this program and check whether your hunch actually works? We'll revisit this problem in detail in Section 4.1 on page 50.

What we'll say for now is that, at the level of abstraction of the C source code, there are no further assignments to `reward` after line 16 and so there should be absolutely no way that the content of that variable is modified after that line: it should stay at either 5 (if line 13 was executed) or 10 (if line 15 was executed). How can the attacker overwrite the `reward` variable after line 16 without another assignment taking place? To even acknowledge that this is possible we must look *below* the abstraction layer provided by the C language and look at what happens at the level of the compiled program. Only then can we understand that it is possible for the attacker to write a long string into one variable and have it overspill into another adjacent variable, somehow breaking the rules of the C language abstraction layer.

And so, even just to *understand* what the attacker is doing, we must now become proficient not only in the programming language we were using to write the application but also in lower level details about how the data structures of our program are allocated in memory, how the high level code statements translate to machine language and so forth. And that's just to explain an attack that some adversary already came up with after *they* discovered a vulnerability. Even greater ingenuity is required for *us* to notice unknown vulnerabilities and anticipate possible new attacks. This is why security is at the same time challenging and fascinating: you are pitting your wits against those of an intelligent opponent; and, to beat that opponent, you must be at least as knowledgeable as them in the areas where this opponent is going to strike—and you don't know in advance what these target areas are. You must be willing to learn a lot, at short notice, about topics that might have nothing to do with your own specialty: assembly language, operating system internals, TCP/IP, firewalls, security protocols, cryptography, digital electronics (even *analog* electronics, for that matter), materials science, locksmithing and so forth. If you have the mindset of an eternal learner, it's great, and it's fun; otherwise it's a chore, and it's very difficult. To outsmart the attackers you have to learn to think like them and anticipate their moves. This requires considerable creativity and lateral thinking.

1.3 Socio-technical systems

When I talk about "systems", I don't mean simply "computer systems" but rather "complex socio-technical systems". Some computer security attacks can be devastatingly effective even without touching the technical part at all, if they target the "socio-" part of the system in the

appropriate way. For example, I once received a worried phone call from someone who needed confidential computer security help about a troublesome and embarrassing incident he was the victim of. This individual, having engaged in sexual self-stimulation at his computer in the privacy of his own home, had received an email from an unknown who told him he had infected his computer with a Trojan, had recorded what he had done through the webcam, and he would send the video to everyone in his address book unless he paid a ransom in Bitcoin. Essentially extortion (a crime much older than computers), but with a technological twist. The victim was in a panic about this and asked me what to do, especially about cleaning his computer from this all-seeing and all-controlling Trojan, as he rightly feared he might get further blackmail requests even if he paid up. Fortunately I was able to reassure him that this was all a bluff: the crook had not installed any Trojan. He had simply crafted a plausible threatening letter and sent it to as many email addresses as he could find[2]. The crook then sat back and simply pocketed the ransom from those who actually believed they had been caught in the act and were sufficiently scared to believe the threats. My correspondent did not initially believe my reassurance that the Trojan was a bluff but he found my explanation convincing once I started reading out to him an essentially identical email that I had also received.

> Greetings!
>
> I have to share bad news with you. Approximately few months ago I have gained access to your devices, which you use for internet browsing. After that, I have started tracking your internet activities.
>
> ...
>
> I have already installed Trojan virus to Operating Systems of all the devices that you use to access your email.
>
> ...
>
> This software provides me with access to all the controllers of your devices (e.g., your microphone, video camera and keyboard). I have downloaded all your information, data, photos, web browsing history to my servers.
>
> ...
>
> Well, I have managed to record a number of your dirty scenes and montaged a few videos...

[2] I knew, because I had received several copies myself. To a first approximation, everyone gets them: it is not a function of the websites you visit but of how many spammers are selling your email address to each other.

> If you have doubts, I can make a few clicks of my mouse and all your videos will be shared to your friends, colleagues and relatives.
>
> ...
>
> Let's settle it this way: You transfer $1600 USD to me (in bitcoin equivalent according to the exchange rate at the moment of funds transfer), and once the transfer is received, I will delete all this dirty stuff right away. After that we will forget about each other. I also promise to deactivate and delete all the harmful software from your devices. Trust me, I keep my word.
>
> ...

There you go. A crafty low-tech attack that targeted a vulnerability not in the computer's software but in the user's psychology. There was no Trojan, just a well thought-out story that would sound believable and persuasive to some victims. Some recipients would know for a fact, as I did, that those events never happened and that the message *had* to be a bluff; but others, oppressed by guilt and shame after believing they had been caught out, might find the story all too plausible and might be scared into paying the ransom. Given the almost-nil cost of sending out the emails to as many addresses as possible, the crook is not worried if the response rate is very low, as it still makes an enormous return on his very modest investment.

There are at least two take-aways from this anecdote: one, that today's computers (and smartphones for that matter) are sufficiently powerful, complex and full of vulnerabilities that the scenario depicted in that email is not too unrealistic. It is fairly believable that a skilled and determined attacker might be able to deploy the kind of Trojan described above. But it would take a rather competent and resourceful attacker: at least at the time of writing (although such things may change over time), this type of hi-tech payload tends to be used sparingly and licensed at high prices to oppressive regimes for the purpose of spearphishing political opponents, dissidents and journalists[3], as opposed to being used by organised crime for dragnet extortion as the email misleadingly suggested. The second take-away, however, is that serious and exploitable vulnerabilities can be found in humans as well as in computers, and that the competent security expert must have a good grasp of human factors in order to be able to protect complex socio-technical systems (which, to

[3]See for example the Pegasus spyware developed by the Israeli NSO group, "used to facilitate human rights violations around the world on a massive scale" (Amnesty international). According to an Amnesty International investigation, it was used against the family of Saudi journalist Jamal Khashoggi both before and after his 2018 murder: https://www.amnesty.org/en/latest/news/2021/07/the-pegasus-project/.

a first approximation, all systems worthy of your attention are). We'll have more to say about human factors in security in Chapter 10 on page 129; but have a look at my entertaining paper with TV star Paul Wilson[4] if you wish to peek ahead.

1.4 Adversarial thinking

To outsmart the attacker you must learn to think like the attacker. To think like the attacker you must learn to spot and exploit vulnerabilities, and thus you must gain some practical experience in attacking systems. This interplay between attack and defense is at the core of security: the two are inextricable. Those who worry that "Help! Help! This irresponsible university professor is teaching the evil terrorists how to attack our homeland!" fail to understand that nobody can ever be a good defender without being skilled at attack. The reason why serious university-level security courses are needed that teach how to attack systems is because otherwise only criminals would learn that craft, from their underground forums. And they already do anyway. What we are doing here is putting the good guys on the same footing. Nobody can build a lock capable of withstanding the unwelcome attention of crooks and burglars without first being a skilled and resourceful lockpicker.

In the cybersecurity world, many adversaries are not particularly clever: we call them *script kiddies*. They are usually very much below the skill level where they could identify a new vulnerability and conceive of an attack to exploit it, let alone the level of being able to write a sophisticated Trojan like Pegasus. But they have become proficient at running pre-packaged attack tools that smarter and more dangerous attackers have created. Script kiddies may not be that smart but they can be quite adept at using those scripts, and they can still run circles around the defenders unless the latter get up to speed.

So this course is not about turning you into script kiddies: it's about making you understand the fundamentals. Things evolve in an arms race, and today's vulnerability might get patched tomorrow, but tomorrow there will be a new one. You need to learn the principles, not the low-level tricks that quickly become obsolete. But, in abstract, the principles alone won't make a lot of sense and won't be of much use. You also need hands-on understanding. You need to see what the bad guys actually do, and where they have to be ingenious, what challenges *they* have to

[4] Frank Stajano and Paul Wilson. "Understanding scam victims: seven principles for systems security." *Commun. ACM* 54, 3 (March 2011), 70–75. DOI:10.1145/1897852.1897872. https://www.cl.cam.ac.uk/~fms27/papers/2011-StajanoWil-scam.pdf.

overcome, before you can have any hope of developing the mindset and the skill set that will allow you to defeat them. Now, as we said, most bad guys are script kiddies and they are not particularly ingenious: they just recycle stuff invented by a small number of *smarter* bad guys. But I want you to become the kind of good guy who can defeat those smart bad guys that are at the top of the pyramid, so you first need to follow in their footsteps. Hence, study the classics. The classics here are attacks that have broken new ground and have made history. Of course they were devastating, so they had to be stopped. Once countermeasures were developed, they gradually became adopted by default. So, in our study of the classics, we will have to go in and disable the countermeasures that are nowadays standard, in order to be able to replicate what the original bad guys did.

We'll repeat this pattern again and again in this course (and specifically in the SEED labs). Wind back the clock to a simpler time when this attack was new[5]. Then recreate the thought process of the attacker, and their action. Then enable the (now standard) countermeasure and see how it stops them. Then see what else the attacker could do at that point. As we said, it's usually a never-ending arms race.

I want to expose you to common failure patterns so that you may recognise them when they occur in new situations, in new computer systems that have not been invented yet but that you will encounter in your professional life. I do not hold in very high esteem those who talk the talk but cannot walk the walk, and I don't want you to be one of them. If you see a vulnerability, you should have the technical skills to exploit it, otherwise you will be pathetically ineffective at stopping those bad guys who would exploit it in anger. My aim is that, by the end of the course, every one of you will have acquired, alongside an understanding of the principles, the skills to exploit a few basic vulnerabilities. To set the right incentive, I'll do my best to set exam questions that can only be solved by people who have acquired those skills. If you want to pass the exam, do the practical exercises. If you attend the lectures and read the handout but without doing the exercises and without referring to the course textbook, you will put yourself at a severe disadvantage.

Of course, with only a 12-hour lecture course, it is unrealistic to expect that you would become an expert, the kind of person who can invent new security primitives, new defenses, new attacks and so on: to become truly proficient at that, as with anything, you need to put in your own proverbial 10,000 hours of practice. This course is meant to offer you a mental roadmap of security, give you a taste for the field,

[5]By disabling a number of countermeasures that are now deployed by default as good standard practice.

and show you the breadth of themes and topics you need to consider. At Cambridge we have a strong research group in security, with a tradition going further back in time[6] than most others at competing institutions; and if you find security attractive we hope you'll join us. But, for now, you're basically just going to study the classics, "the literature" as it were, and learn the most basic things that crackers and criminals have been doing for decades. Not everything you encounter in your professional life will be as basic as what I show you in this course, and indeed many modern systems will have been patched against the well-known attacks we are going to explore, but nonetheless the same patterns of failure tend to reoccur in new guises. Besides, you'd be surprised and appalled at how many deployed systems are still unpatched against decades-old vulnerabilities. So, even if studying those classics is only a first baby step, it is highly educational and it will take you quite a bit further than you might think.

When speaking to intelligent and responsible adults I should not have to say the following, but I'll say it anyway, and I'll even put a box around it and stick it at the front of the book: Stay Legal. I'll be teaching you potentially dangerous stuff and it's your responsibility to use it for good and to make sure that you don't accidentally hurt anyone, including yourself.

The core of security is adversarial thinking. It is a mindset. It is difficult, perhaps impossible, to teach. All I can do is inspire you to want to pick it up. To develop it and refine it, you need experience. How can you get any? There are many avenues. The main one is the SEED labs: they do require a lot of work but they are an integral part of this course. Do not skip them merely because they do not attract marks: you will never own this material without doing the exercises. Practice as much as you can. Engaging in CTFs is always good, and I have actively participated in the organisation of such events for university students for many years because I consider them highly beneficial. Take part in them and learn from your peers. I occasionally invite the most talented among you, including medalists from past CTFs, to do small, self-contained and well-paid jobs for my security company, Cambridge Cyber. A typical task might be to perform penetration testing on a device or a system belonging to my client. You need a good mix of creativity, lateral thinking, experience, deviousness, honesty and intelligence. The answer won't be in any textbook, but it will typically be accessible to

[6] For example, in the 1960s, before even the invention of public key cryptography, Roger Needham, the PhD supervisor of my PhD supervisor, invented the technique whereby passwords are stored in hashed form on the server. This idea was picked up by Unix in the 1970s and is now almost universally adopted.

Stay legal. Don't break anything.

In this course I am going to teach you some attacks and I will encourage you to become proficient at them. These attacks can cause serious damage: cyber weapons aimed at critical infrastructure use many of these same techniques. *Do not launch these attacks on any system other than your own experimental setup.* Understand how to confine your experiments and do so. Ensure that no other system is damaged or affected by your experiments. Stay legal at all times. You will be liable to be sued or prosecuted otherwise, and claiming that you did it for coursework will never be accepted as a mitigating excuse.

you if you know the textbook inside and out. Please make the most of all these opportunities to get hands-on experience. The few who choose not to engage in any practical work (particularly the easily accessible SEED labs) are cheating themselves out of a valuable education, and will have a hard time passing the exam for this course at the end of the year.

You must learn adversarial thinking to anticipate what the bad guys might do. You have to look at everything with suspicion. You must question all assumptions. That's in part what makes security interesting, and appealing, and fun, to those of us who are contrarians at heart and are reluctant to accept any dogma. But, as I told you, you have to be prepared to learn a lot more than is usually required in a regular course, because all the assumptions that you might end up questioning are there for a reason: they are there because the world is too complex and we need to simplify it. And the simplification is correct most of the time but slightly wrong some of the time, and the clever attacker exploits that subtle discrepancy between assumption and reality. And so, to follow the attacker—or, even better, to *anticipate* the attacker—you must be well versed not just in the easy simplifying assumption but also in the rather more complicated reality. Abstraction is one of the most powerful tools in the computer scientist's arsenal to dominate the inherent complexity of the field, but in the security game you must be prepared to abandon the abstraction and go back to first principles.

Going down that route can be difficult, especially when you are the first one to do it and nobody has yet pointed out to you which abstraction needs to be questioned and unravelled. But that's why good security people are so valuable, and in great demand. It's useful if you are able to recognise a pattern, in your client's security problem, that is similar to something I've shown you in this course, or that you've seen elsewhere in your extra-curricular security experience. It will rarely be *exactly* the same; on the other hand, many core ideas resurface in different guises. When you do find a match, and this inspires you to develop and apply the relevant solution or countermeasure, it's really satisfying. It's also really valuable, because security problems cost money to your client, and you'll be saving them this money. And a secure and satisfied client will be happy to pay you a nice fraction of what you saved them.

Chapter 2

Access control

Textbook

Study Chapter 1 of W. Du, *Computer Security* 3rd ed.

Since you are taking the trouble to read this, I shall assume that you diligently studied and enjoyed the Unix Tools course, as opposed to skipping it because it was not examinable. If you took the lazy shortcut then, go back and pay your debt now: Unix Tools is a prerequisite for this course.

If you have practical experience with the Unix command line, you may already be familiar with most of the material in this chapter. Great. If you believe this to be the case and have better things to do with your time, you may skip ahead to Chapter 3 on page 41, provided that you are able to answer all of the following questions correctly. The questions are answered implicitly in the chapter[1], although the last one requires a little extra lateral thinking and man page browsing. You may self-verify your answers by trying them out on your Unix box, or on your SEED VM if you don't normally use one. If any of this sounds difficult, read on. You need this knowledge as background for Chapter 3.

1. Explain in detail what each of these commands does, where `blob` is a file and `folder` is a directory.

   (a) `chmod 327 blob`

   (b) `chmod 1777 folder`

[1] Why no explicit solution notes? If you truly know the answer, you will already know that you do. If you are just guessing, it doesn't matter whether your guess is right or wrong and you should learn this topic properly anyway.

(c) `chmod u+s blob`

(d) `umask 27`

2. User `alice` is in group `staff` but not in group `seed`. For each of the following 8 files (`yen`, `banana`, `grape`, `kiwi`, `cat`, `dog`, `bike`, `skateboard`), state whether she can read, write and/or execute it, and why.

```
$ ls -ld {f,p,v,c}* {f,p,v,c}*/*
drwxrwxr-x 2 seed  seed  4096 Mar 30 20:50 currencies
-rw-rw-r-- 1 seed  seed   318 Mar 30 20:50 currencies/yen
d--xr-xr-x 2 alice staff 4096 Mar 30 20:30 fruits
-rw-rwxr-x 1 seed  seed   160 Mar 30 20:30 fruits/banana
----rw-r-- 1 seed  seed   196 Mar 30 20:30 fruits/grape
----r----- 1 seed  seed   497 Mar 30 20:30 fruits/kiwi
drwxr----- 2 seed  staff 4096 Mar 30 20:40 pets
-rw-rw-r-- 1 seed  seed   264 Mar 30 20:39 pets/cat
-rw-rwxr-x 1 seed  seed   792 Mar 30 20:40 pets/dog
dr--rwxr-x 2 alice staff 4096 Mar 30 20:39 vehicles
-r--rw-r-- 1 seed  seed   528 Mar 30 20:39 vehicles/bike
-rw-rw---- 1 seed  seed   264 Mar 30 20:39 vehicles/skateboard
```

3. What exactly should Ulrika do to gain root privileges, in the scenario described in Section 2.1 on page 29? Demonstrate the attack on your SEED VM.

4. Again, user `alice` is in group `staff` but not in group `seed`. User `seed`, in group `seed`, has just executed the following commands. What commands should Alice issue to display the content of the `q4/secret.txt` file?[2] Demonstrate the attack on your SEED VM.

```
(seed) $ cp /usr/bin/find .
(seed) $ chmod 4555 ./find
(seed) $ ls -ld * */*
-r-sr-xr-x 1 seed seed 320160 Mar 31 09:15 find
drwxrw---- 2 seed seed   4096 Mar 31 09:11 q4
---------- 1 seed seed     23 Mar 31 09:11 q4/secret.txt
```

2.1 Multi-user systems

At the processor level, the machine does not have a concept of "user" or "account": all the processor sees is machine code instructions, and it just

[2] Bonus marks if she can do that without leaving any obvious signs of her break-in.

executes those instructions. Users and accounts are operating-system-level concepts. The first personal computers, taking as the prototypical example the IBM PC of 1981, were single-user machines, as the "*personal*" qualifier suggested. The MS-DOS operating system that ran on the IBM PC and its clones had no concept of users: there was a single unnamed user, sitting at the console of the machine. When the machine started, it did *not* ask the user to log in: once the boot sequence completed, the user would simply be greeted by the DOS command line prompt. Any program the user ran was allowed to do anything it wanted with the machine, its peripherals and its files. The payload of a DOS virus could easily reformat the whole hard disk, deleting everything.

The much larger computers used by large corporations did, of course, support multiple users[3] and eventually the ability for one machine to support multiple users migrated to the operating systems of home and personal computers as well, largely driven by the rise of the Internet and by the fact that email and then the web spread to personal machines.

If an operating system offers accounts to multiple users on the same machine, one of its tasks is to safeguard those users from mutual interference. User Alice must be guaranteed that user Bob will not be able to read, change or delete any of her files unless she explicitly gives him permission to do so. However, as we said, the processor itself has no concept of users and accounts. The high level instruction to "read the file with Alice's patentable invention blueprint" translates to lots of low level instructions such as "read the disk block at this address and stick it into memory at that address". If the processor is currently executing a program that was launched by Bob, what stops Bob's program from issuing those same machine instructions, and thereby reading Alice's secret invention? The operating system, who knows about Alice and Bob and about Alice's wish to keep her patentable blueprint to herself, could in theory block the action, but we don't want the operating system to have to check every individual machine code instruction for legitimacy, otherwise the system would be monstrously slow. We need a mechanism whereby Bob's machine code instructions can run at full speed, and yet Bob is not allowed to access any of Alice's resources unless Alice granted him permission to do so.

One sensible way of doing that is to ensure that only the operating system may access the files[4]. Whenever user programs also wish to access files, they have to ask the operating system to do so on their behalf.

[3] If nothing else because you could not give an individual computer to each employee when the computer itself was too large to fit in the employee's office.

[4] Or I/O, or more generally any resource that might be object of contention between users.

Nice idea, but what's there to stop Bob's program from issuing the same machine code instructions that the operating system issues for accessing Alice's files? After all, machine code is machine code, as far as the processor is concerned—it does not come with provenance indicators. If we go for this arrangement, we need some facility for the processor to know whether it's running the operating system or a user program—essentially one special bit inside the processor corresponding to the predicate "is this instruction going to be executed by the operating system or by a user process?", so that the processor may decide whether to allow it (for regular instructions) or disallow it (for I/O or otherwise privileged instructions if invoked by a user process). The Intel 8088 processor in the original IBM PC did not have such a bit: there was no notion of it being in real mode or in supervisor mode, and so MS-DOS could not have been multi-user even if it had wanted to (which it didn't anyway). Subsequent processors in the x86 family, starting with the 80286, introduced a supervisor mode bit and then fancier facilities with a variety of privilege levels. But the intellectual nugget here is just that one bit: the idea that the hardware is able to tell whether it's running the operating system or a user process. Then the OS has all it needs to impose limitations on what other processes can do, without the computing system having to incur a performance hit. Once that hardware bit is there, all the rest can be done in software. Conversely, if that hardware bit is not there, no amount of sophistication in the OS can compensate for its absence, short of analysing each instruction in software before execution, which would unacceptably degrade performance.

Once we have this kind of arrangement, with the operating system running at a higher privilege level than user processes, this sets the scene for most of the battles we are going to study in the realm of software security. Access to the "precious" resources of the machine is mediated by the OS, which is "more trusted" than the user-level programs, in the sense of being allowed to do more dangerous things. The following insightful but seemingly paradoxical definition of trust is due to Bob Morris[5].

> A trusted entity is one that can violate your security policy.

[5]Computer security demigod Bob Morris (1932–2011) was initially a researcher at Bell Labs, where he worked among other things on Unix security, and later became the Chief Scientist at the NSA. He also happened to be the father of Robert Tappan Morris (1965–) who, in 1988, aged 22, created the Morris Worm (cfr. Section 11.1 on page 145), one of the first and most influential pieces of self-spreading Internet malware.

The user may run less trusted programs, including programs of unknown provenance that might actually be malicious, but these will not be able to access protected resources unless the OS grants them permission to do so. In this context, the typical security battle is *privilege escalation*: the malicious user-level program wants to execute code with the same privileges as the OS. Once this status is reached, the adversary has won, because no restrictions are imposed on the OS, which is axiomatically trusted. After attaining the privilege level of the operating system ("getting root"), the adversary will be able to do anything that the machine can physically do, including accessing all files[6].

According to the *principle of least privilege*[7], even those who are allowed to log in as root should use that privilege only when strictly necessary and perform most of their work from their unprivileged personal account, rather than remaining logged into the root account all day. Otherwise they might, among other things, accidentally run malicious programs using their root credentials. For example, if naïve system administrator Roger were in the habit of doing all his work as root (bad idea), and also had the current directory in the path[8] and ahead of the others (another bad idea), what could naughty user Ulrika do to obtain root privileges for herself? (Remember that any program launched by root is executing with the privileges of the root account.) Think about it for a moment, come up with your own full solution (extra credit if you even try it out in your SEED VM and it works), and let's get back to this problem once we have discussed a few technical details, in particular setuid (Section 2.3.4 on page 36).

With this as the background, in this chapter we focus on defining which users can access which files. We refer to Unix-like file systems, but only in order to illustrate fundamental security ideas with concrete

[6]The confidentiality of the data in the files might still be protected by strong encryption, provided the decryption keys have not been stored in RAM or in other places accessible to an attacker with root privileges.

[7]"Every program and every privileged user of the system should operate using the least amount of privilege necessary to complete the job.". Jerome Saltzer, "Protection and the control of information sharing in multics". *Comms ACM* 17(7):388–402, 1974. https://dl.acm.org/doi/10.1145/361011.361067

[8]"The path" is a colon-separated list of directories where the shell looks for executables whenever the name of a program is typed on the command line. It is stored in the aptly-named `PATH` environment variable, as in

```
echo $PATH
/sbin:/usr/bin:/usr/local/bin
```

If I were to type `grep` at the command line, the shell would look for an executable program by that name in `/sbin`, then in `/usr/bin` and so forth until it found it, and it would then execute that first instance it found.

examples. This course does not attempt to cover security details for all current platforms and, even for the narrower case of POSIX, it does not attempt to cover all the details. Those of you who wish to become competent sysadmins will have to read up (and practice) a lot more.

2.2 Discretionary vs mandatory access control

You may have already been introduced to multi-level security and to the Bell–LaPadula security policy model, in which each object is assigned a classification level reflecting its confidentiality[9] and each process is only allowed to access objects up to a designated level. Given that a process may read from some object and write to some other object, the following concern must be addressed: secret information could be read from a high level object by a process legitimately cleared to high level, and it might then be leaked into a low-level object through a write performed by that high level process. Metaphorically, the four-star general is cleared to read the "Top Secret" battle plans, but he might leak some of these details if he ever writes a merely "restricted" report[10]. The Bell–LaPadula policy aims to prevent such occurrences by enforcing the two rules of "no read up" (obvious: the cleaner may not read the battle plans) and "no write down" (more subtle: the general may not speak to journalists).

Such access control restrictions are called *mandatory* because they apply unconditionally to every read or write operation. The operating system intercepts all reads and writes and, after checking the classification level of the corresponding process and object, it only executes those read and write requests that comply with the BLP rules. The user, whether cleaner or general, has no discretion in the matter, and no power to override those rules.

In contrast, *discretionary* access control refers to the situation whereby it is up to the user to decide what access control restrictions, if any, should be imposed on the files owned by that user. Discretionary access control is what you are used to in the POSIX file system (or NTFS for that

[9] The exact hierarchy of levels is country-specific but might typically range from "Unclassified" to "Official", "Restricted", "Confidential", "Secret" all the way up to "Top Secret".

[10] For a case of "life imitating art" in terms of violations of Bell-LaPadula, recall Donald Trump, during his first term in office as US President, repeatedly disclosing classified information to journalists, to Russian diplomats, or on Twitter, as well as retaining boxes of classified information, including nuclear secrets, in his private Mar-a-Lago residence. https://en.wikipedia.org/wiki/Donald_Trump%27s_disclosures_of_classified_information

matter), and what we'll be primarily looking at in the next few sections.

The general view of discretionary access control is that, if you arranged all the users and all the files of the system into a matrix (Figure 2.1 on page 31), with one column per user and one row per file, you could specify in each cell of the matrix what access rights should be granted to that user on that file, such as "this user may read (and/or write, and/or execute) this file".

Splitting the matrix into rows tells you who may access a given file ("access control list" view), whereas splitting it into columns tells you which files a given user may access ("capabilities" view). This solution offers maximal granularity but in general the resulting matrix is too large to be manageable, regardless of whether you split it by rows or by columns. Practical systems, including the POSIX model we'll be looking at next, replace it with compromises (such as the introduction of *groups* of users) that offer a little less flexibility while still addressing most of the common cases.

2.3 POSIX file system security

2.3.1 Users, groups, files and directories

Access control in a POSIX file system is based on the following model.

The operating system supports **users**[11], who have individually-defined rights to access various resources[12] on the system. The **root** user, also known as the **superuser**, with a numeric user id of 0, is privileged and corresponds to the administrator role: it is not subject to the restrictions imposed on the other users and it may, among other things, create new users and impersonate any user.

Each **file** has a unique **owner** (a user) and is associated with a unique **group** (a named set of users). Each user belongs to one or more groups, one of which is designated as the **primary group** for that user: files created by that user are assigned to the user's primary group on creation.

[11] Internally, each user account is identified by an integer. A mapping between those integers and human-readable login names is kept in the `/etc/passwd` file.

[12] In particular files, but then Unix treats almost anything as a file.

	alice	bob	charlie	root
/usr/bin/ls	r-x	r-x	r-x	rwx
/home/bob/note.txt	---	rw-	rw-	rw-
/home/alice/bin/calc	rwx	---	r-x	rwx

Figure 2.1: Access control matrix

The file system is arranged hierarchically into a tree of directories nested to arbitrary levels. A **directory** is a structure that may contain files or directories. Each file resides in a specific location in the tree. Each location in the tree corresponds to a path, which is a possibly empty list of directory names[13].

Each file is associated with some metadata, which is stored in a separate data structure called the **inode** alongside pointers to the actual data of the file[14]. A directory is a special kind of file that maps human-readable file names[15] to inodes[16]. Among the metadata stored in the inode are the **permission bits**: a triplet of triplets indicating the ability to read, write or execute the file for each of user, group and other. These permission bits are displayed by the "`ls -l`" command as a 10-character string similar to `drwxrwxr-x`, where a letter indicates that the corresponding bit is set to 1, and a dash indicates that it is set to 0. The three `rwx` triplets for `u`, `g` and `o` respectively are preceded by a `d` bit that is set whenever the entry is a directory rather than a regular file.

2.3.2 Permission bits and `chmod`

Permission bits are used by the OS in the following way. A process p, launched by user u, requests an action a (read, write or execute) on a file f. Should the OS grant or deny permission? If u is root, then permission is granted regardless of any other considerations. Otherwise, the OS first determines which of the three triplets should apply (one and only one will): if user u is the owner of file f, the `u` triplet applies; if u is not the owner but belongs to the group of file f, the `g` triplet applies; otherwise, the `o` triplet applies. Once the correct triplet has been established, its `r`, `w` and `x` bits say which of the corresponding actions are allowed.

[13] The forward slash character (`/`) is used to denote the root of the tree and to separate the directory names in a path.

[14] Different POSIX-compliant file systems use different low-level data structures to address the individual blocks that make up the file, with various efficiency-related trade-offs. We are not describing these details here.

[15] "File names are infinite in length, where infinity is set to 255 characters." (Peter Collinson, *The Unix File System*) This quote is slightly dated, though, and some modern POSIX systems set infinity to a higher value.

[16] It is possible for several file names to point to the same inode: we then speak of "hard links". The inode contains a reference count. When several files hard-link to the same inode, none of them is special or privileged—they are all equivalent and independent ways of accessing the file's data. The file deletion operation is implemented through the `unlink()` function, which in fact only deletes the mapping between that file name and that inode, decreasing the reference count by one. Only when no more file names refer to a given inode (i.e. when the reference count reaches zero) are the corresponding data blocks actually deleted.

For directories, the semantics of the permission bits are slightly different. The `r` permission means you may read the directory and thus find out the names of the files it contains. The `w` permission means you may write to the directory and thus add or remove files. The `x` permission, which clearly can't mean "execute the directory", is the permission to navigate into the directory (make it the current directory of the process) or to open the files and directories therein.

The `chmod` command changes the mode (permissions) of a file. In the symbolic form you specify which triplet to affect (`u`, `g`, `o` or combinations, or `a` for "all of them"), then whether you want to add permissions (`+`), remove them (`-`) or set them regardless of previous values (`=`), and then the permissions in question (`r`, `w` or `x`, or combinations thereof). Finally, the list of files whose modes you wish to modify. See examples below.

Note that, if you have `--x` access to a directory, you may not list the files it contains, but you may open any files or directories in it of which you already know (or can guess) the name. Conversely, if you have `r--` access to a directory, you may see the names of the files it contains but you may not access those files (nor even produce a full `ls -l` directory listing for them, because doing so requires accessing their inodes to read the metadata such as file sizes and permissions).

```
$ ls -ld folder/ folder/*
drwxrwxr-x 2 seed seed 4096 Mar 30 19:19 folder/
-rw-rw-r-- 1 seed seed   33 Mar 30 19:19 folder/limerick.txt
$ cat folder/limerick.txt
there was a young lady from Riga
$ chmod a=x folder/
$ ls -ld folder/ folder/*
ls: cannot access 'folder/*': No such file or directory
d--x--x--x 2 seed seed 4096 Mar 30 19:19  folder/
$ ls -l folder/limerick.txt
-rw-rw-r-- 1 seed seed 33 Mar 30 19:19 folder/limerick.txt
$ cat folder/limerick.txt
there was a young lady from Riga
$ chmod a=r folder/
$ ls -ld folder/
dr--r--r-- 2 seed seed 4096 Mar 30 19:19 folder/
$ ls -l folder/*
ls: cannot access 'folder/limerick.txt': Permission denied
$ ls -l folder/limerick.txt
ls: cannot access 'folder/limerick.txt': Permission denied
$ cat folder/limerick.txt
cat: folder/limerick.txt: Permission denied
```

Note the counterintuitive fact that, if the file permission bits grant some permissions to the group but not to the owner, then if u is both the owner and a member of the file's group she will be denied access, even though one might think that her membership of the group would grant her the right. It doesn't. Only one of the triplets applies (the "topmost" one). This state of affairs catches many people by surprise because it occurs infrequently in practice: it is common and natural for `u` to have at least as many privileges as `g`, and for `o` to have the least privileges of all. But it is not mandatory.

```
(seed) $ whoami
seed
(seed) $ groups
seed adm cdrom sudo dip plugdev lpadmin lxd sambashare docker
(seed) $ ls -ld folder/
drwxr-xr-- 2 seed cdrom 4096 Mar 30 19:47 folder/
(seed) $ ls -l folder/weird-permissions.txt
--w-rw-r-- 1 seed cdrom 26 Mar 30 19:47 folder/weird-permissions.txt
(seed) $ cat folder/weird-permissions.txt
cat: folder/weird-permissions.txt: Permission denied
(seed) $
(seed) $ sudo su alice
(alice) $ whoami
alice
(alice) $ groups
alice cdrom
(alice) $ ls -ld folder/
drwxr-xr-- 2 seed cdrom 4096 Mar 30 19:47 folder/
(alice) $ ls -ld folder/weird-permissions.txt
--w-rw-r-- 1 seed cdrom 26 Mar 30 19:47 folder/weird-permissions.txt
(alice) $ cat folder/weird-permissions.txt
I like to move it move it
```

Because file permission bits are grouped in triplets, when expressing them numerically it is convenient to do so in octal (base-8) rather than decimal or hexadecimal, because one octal digit, from 0 to 7, is worth exactly 3 binary digits. Thus `rwx` is 111_2 or 7_8, `rw-` is 110_2 or 6_8, `r-x` is 101_2 or 5_8 and so forth. The three triplets for `u`, `g` and `o` become three octal digits: `rwxr-xr-x` is 755, `rw-r--r--` is 644 and so on[17]. Note that, unlike the more readable and more expressive symbolic form that lets you set or reset specific bits without touching the others, the more compact

[17] Of course we don't type the subscript $_8$ in the argument of the `chmod` command on the POSIX command line.

octal form only lets you use the absolute form of `chmod` where you set all the permission bits at once, regardless of their previous values.

```
$ ls -l chapter.tex
-rw-rw-r-- 1 seed seed 177 Mar 30 19:13 chapter.tex
$ chmod 640 chapter.tex
$ ls -l chapter.tex
-rw-r----- 1 seed seed 177 Mar 30 19:13 chapter.tex
$ chmod o+w,u-w chapter.tex
$ ls -l chapter.tex
-r--r---w- 1 seed seed 177 Mar 30 19:13 chapter.tex
$ chmod a=rw chapter.tex
$ ls -l chapter.tex
-rw-rw-rw- 1 seed seed 177 Mar 30 19:13 chapter.tex
```

2.3.3 Umask

What are the permissions of newly created files? By default, all possible permissions are granted (666 for data files and 777 for executables), but you may restrict that liberal default by defining a umask in an appropriate startup file. Every process inherits a umask from its parent. The umask is an octal pattern describing the permissions you do *not* wish to be granted by default; or, in other words, the permissions that should be "subtracted" (more accurately: masked out) from the default.

If for example you wanted everyone to be able to read your files, but only you and the group to be able to write and execute them[18], you would remove the write and execute permissions from "others" by setting your umask to 003, where $3_8 = 011_2 = 010_2 + 001_2$ (write-permission OR execute-permission). If instead you wanted to extend that same restriction (no write nor execute) to the group as well, you would set your umask to 033.

Any newly created file would thus acquire the permissions of *default* AND NOT *umask*. In our first example above that would be 777 AND NOT 003 = 774 = `rwxrwxr--` for executables and 666 AND NOT 003 = 664 = `rw-rw-r--` for data files. In our second example we would have 777 AND NOT 033 = 744 = `rwxr--r--` for executables and 666 AND NOT 033 = 644 = `rw-r--r--` for data files.

A common default value for umask is 022, which removes write permission from group and others.

[18] In other words, if you wanted your regular files to be 664 and your executables 774.

2.3.4 Setuid

There are situations in which users need some specific form of access to certain protected resources but it is undesirable to give them *unrestricted* access to such resources. For example, if the system keeps the login credentials of the users in a file[19], and users are allowed to update their login credentials, then user Alice should be allowed to write to that file when she changes her password. Disallowing that would be a pain, because the administrator would need to edit the file on her behalf every time she changed her password; but, on the other hand, it would be a bad idea to grant her unrestricted write access to the whole file because that would allow her to change Bob's password too. How nice it would be if we could say "Alice is allowed to change this file, but only *her own line* in this file, and in fact *only the second field* of her line, because we don't want her to mess up the rest". Clearly these semantics are too complex and fine-grained to fit within the simple model of the `ugo` and `rwx` permission bits.

A clever solution to this problem was invented in the early 1970s by Unix pioneer and 1983 Turing Award laureate Dennis Ritchie (1941–2011): the *setuid* bit. If an executable file is tagged with the setuid bit, when it is launched the operating system pretends to itself that the process was started by the owner of the file, rather than by the user who actually launched the program. This allows an unprivileged user to invoke a privileged action, but the privileges that the user may exercise are strictly constrained to the exact behaviour that was compiled into the setuid-endowed program. In the example above, Alice, who does not have write access to the `/etc/shadow` file containing the salted hash of her password, is given permission to execute the setuid-root program `/usr/bin/passwd`. Since the root account has write access to `/etc/shadow`, the `/usr/bin/passwd` program running as root has permission to edit that privileged file, but its carefully crafted code will constrain Alice to modifying only her own entry, and will force her to preserve the correct format for it while doing so.

At the operating system level, each process contains three numeric UID (user id) fields in the process descriptor table, the values of which are not necessarily distinct:

Real user id (`ruid`): the UID of the user who launched the program.

[19] Which POSIX does. Historically, the information needed to verify user passwords on login was kept in `/etc/passwd`, but it was later moved to `/etc/shadow`, for reasons that will be discussed in Section 7 on page 90. Note that, as we'll explain then, `/etc/shadow` contains not the actual passwords but the salted hashes of the users' passwords.

Effective user id (`euid`): the UID under which the program is currently running, and whose permissions are checked when making access control decisions.

Saved user id (`suid`): a UID that the process previously had and to which it is allowed to revert when desired.

From the man page of `setresuid()`: "An unprivileged process may change its real UID, effective UID, and saved set-user-ID, each to one of: the current real UID, the current effective UID or the current saved set-user-ID".

The `ruid` and `euid` are pretty obvious once the concept of setuid has been understood, but `suid` is sometimes confusing. Why wouldn't the first two be sufficient? If the setuid-root program wants to drop privileges, why doesn't it simply revert to the real UID? Why does it need the saved UID at all? The answer is that yes, the `suid` is not necessary for dropping privileges, but it is needed to re-acquire the elevated privileges after dropping them. Take a setuid-root program launched by unprivileged user Uli. On launch, the `ruid` is Uli and the `euid` is root. The process may drop privileges by setting the `euid` to be the `ruid`, but at that point it would no longer be able to restore root privileges if it had not saved the `euid` into the `suid` before dropping them.

At this point we may get back to our characters Roger and Ulrika from page 29, as we have now covered all the technical pieces we need for Ulrika to carry out her devious plan. She writes a shell script that makes a copy of `/bin/sh` to some secret directory, changes its owner to root[20] and sets the setuid bit[21]. (Note that, if she tried to execute that script herself, it would fail, because she does not have the necessary permissions. But this does not stop her from writing the script, without executing it.) Now she needs to trick Roger into executing this script (as root). She renames it as `ls` and puts it into her home directory. She adds code at the end of the script that actually calls the real `/bin/ls` and also, for covertness, a "self-destruct" line that deletes the script after it has executed. With a dash of social engineering she then asks Roger for help about something strange going on in her home directory. For good measure, she makes her home directory `drwx------`, so that Roger won't be able to see inside it under his own (unprivileged) user id. As soon as Roger, as root, changes into Ulrika's directory and types `ls` to see what's up, Ulrika's booby trap springs shut and she squirrels away a setuid-root copy of `/bin/sh` that gives her root privileges. Game over.

[20] Only root can do that.

[21] Only root or the owner of the file can do that.

I highly recommend trying all this out in your SEED virtual machine. See if you can write Ulrika's script and make it all work as intended. The devil is in the details, and those of you who put in the effort to master these details will be rewarded at the exam (and, more importantly, in their computing career).

It would also be a good idea for you to figure out what advice to give Roger, besides not adding the current directory to his path, and specifically what a more cautious sysadmin should have done in response to this user support request.

We note in passing that the setuid idea also applies to the group: there is indeed also a setgid bit which, if set, causes the process to run under the effective group id of the file's group, as opposed to the primary group id of the user who launched the executable.

The setuid and setgid bits are stored in the inode of the file in a fourth triplet[22] alongside the three triplets of `rwx` bits, for a total of 12 permission bits. The `chmod` command mentioned above in Section 2.3.2 on page 33 accepts numeric modes with four octal digits, where the most significant one is the concatenation of the setuid, setgid and sticky bits.

In the symbolic version, the setuid (4000_8) or setgid (2000_8) permission is indicated with `s` (and the sticky bit with `t`), as in `chmod u+s prog` to make `prog` become setuid, or `chmod g+s prog` to make `prog` become setgid. However, in the output (of `ls -l` for example), in order to save space, no additional character positions are allocated to those topmost three bits. Instead, setuid and setgid are indicated by replacing the `x` with an `s` in the `u` or `g` triplet respectively (or with a `t` in the `o` triplet for the sticky bit). The `s` or `t` is lowercase if the corresponding executable bit is also set (as would be sensible), but capitalised if not.

2.3.5 Sticky bit

The sticky bit is a somewhat obscure facility that we mention here more for completeness than because it is particularly significant.

Now that terabytes are so cheap, it might be hard to imagine times when kilobytes were expensive; but it is in those times that the sticky bit was introduced. RAM was scarce and unable to hold all of the running programs and their data simultaneously; processes would be swapped out while not running, which of course had high I/O costs and greatly slowed down the machine. The sticky bit was used[23] to signal to the OS that a certain frequently used executable was likely to be used again soon

[22] Together with the "sticky bit" covered in the next section, 2.3.5.

[23] In a cooperative setting—you may imagine how this would lead to chaos and tragedy of the commons in a competitive/adversarial setting.

and should therefore be kept in memory even when not running (space permitting) rather than being swapped out. These semantics are now obsolete and no longer relevant.

Since the space for the sticky bit in the inode had by then been reserved, the sticky bit was eventually repurposed—not for executable files but for directories. Normally, to create or delete a file in a directory, a user needs write access to the directory. But consider a shared directory, such as `/tmp`. If, in order to allow several users to create files in there, we grant them write permission to the directory, the undesirable side effect is that they may now delete each other's files. The sticky bit, when applied to the directory, stipulates that, to delete, overwrite or rename a file in that directory, a user must not only have write access to the directory but must also be the owner of the file (or of the directory, or be root). With this arrangement the various users may share the directory without the danger that they might delete each other's files.

Chapter 3

Privilege escalation

Textbook

Study Chapter 2 and optionally 3 in W. Du, *Computer Security* 3rd ed.

SEED labs

Complete the following lab: Software Security | Environment Variable and Set-UID Lab (tasks 6, 7, 8; other tasks optional).

The main high level idea of this chapter is that *any* input of a program (and there are usually more inputs than might seem at first sight) could be exploited by an attacker to make the program behave in unintended ways. If the program is setuid, the exploits typically aim at privilege escalation[1]. Exploitation tends to happen because the author of the vulnerable program did not anticipate all the possible inputs that a malicious user might provide, and failed to sanitise the received inputs to limit them to a safe subset—or failed to quote the inputs to ensure they would only be interpreted as data of the intended type.

Try answering the following questions on your own. If you don't know about a particular Unix program or library function, consult its man page. If there are questions you cannot solve, study the textbook and the rest of this section. In any case, I highly recommend you do the indicated SEED lab to confirm that, as a genuine computer science

[1]Privilege escalation, as we first mentioned in Section 2.1 on page 29, is when a process with limited privileges exploits a vulnerability in order to acquire additional privileges.

student, you can actually *do* the stuff, rather than merely handwave about it like a humanities student or a salesperson might. ☺

1. How can you list all the setuid root programs that are present on your system? Give a specific command, including all its parameters. (Hint: `man find`)

2. What does `/usr/bin/chsh` do? Why does it have to be setuid root? If it did not sanitise its inputs, how could a malicious user exploit it to gain root privileges?[2]

3. How could you exploit the following program, assumed to be setuid root, to gain root privileges? How would you fix the vulnerability?

```
#include<stdlib.h> // system()
void main() {
  system("ls");
}
```

4. How could you exploit the following program, assumed to be setuid root, to gain root privileges? How would you fix the vulnerability?

```
#include<stdio.h> // sprintf()
#include<stdlib.h> // system()
void main(int argc, char* argv[]) { // requires one argument
  char command[80];
  sprintf(command, "/usr/bin/cat %s", argv[1]);
  system(command);
}
```

3.1 Attack surface

> "The Attack Surface describes all of the different points where an attacker could get into a system, and where they could get data out."
>
> (OWASP)

[2] This vulnerability actually existed in the original `chsh` but the current version found in the SEED lab VM does sanitise its input, so you won't be able to test the vulnerability there.

3.1.1 Command line parameters

When a program accepts command line parameters, these are obviously user input. The `chsh` program in question 2 on page 42 above allows a user to change her login shell. The entry for Alice in `/etc/passwd` might look as follows:

```
alice:x:1002:1002:Alice,,,:/home/alice:/bin/bash
```

The line is a record with 7 fields separated by colons. The first field is the username and the seventh is the shell for that user. The `/etc/passwd` file is only writable by root but the `chsh` command allows non-root users to modify their own line (or more precisely just the seventh field of their own line). The program must be setuid root in order to do that. All is fine when Alice plays by the rules and types something like

```
chsh -s /bin/zsh
```

which changes the above line in the file to

```
alice:x:1002:1002:Alice,,,:/home/alice:/bin/zsh
```

But what if Alice supplies a two-line input[3] instead of `/bin/zsh`? The original programmer of `chsh` never imagined that one would enter a multiline input at this stage and thus never sought to prevent that. But that's how Alice, an unprivileged user, might add an extra line (an extra user account) to `/etc/passwd`. If the numeric UID of that account were 0, it would even be a root account[4]. So this is an example of privilege escalation achieved by passing a command line parameter of unexpected format to a setuid root program. One way to block this attack is for `chsh` to sanitise its input by ensuring that the supplied parameter does not contain newlines. Even better practice is to restrict the supplied parameter to be only one of a small number of whitelisted valid choices, namely the entries in `/etc/shells`.

3.1.2 Environment variables

The command line parameters are the most evident portion of the attack surface of a command line program. A less obvious form of input is the environment variables that change the behaviour of the program. The

[3] How can she pass a string containing a newline as a command line parameter without the newline completing her command to the shell?

[4] If Alice wanted to assign a password known to her to this new account, how would she cope with the fact that this method does not let her edit the `/etc/shadow` file as well?

`PATH` environment variable tells the shell where to look for the files it is asked to execute. In the scenario of question 3 on page 42 above, by placing one of her own directories in front of `/bin` in the `PATH`, and by placing a copy of `/bin/sh` in that directory, but renamed as `ls`, Alice will cause `system()` to pick up her fake version of `ls`, which is in fact a shell, and therefore to give her a root shell (since the `ls` program was launched from a setuid root program). This is an instance of privilege escalation by manipulating an environment variable, which is an indirect input to our target program. One way to prevent this attack is by specifying a full, absolute path for the executable to be launched.

This is the countermeasure adopted in question 4. An absolute path for the `cat` executable is supplied to the setuid root program: this makes it pointless for the attacker to manipulate the `PATH` variable in the hope that the victim will execute the attacker's malicious version of `cat`. However, since the setuid root program invokes `system()`, which accepts a string containing the program and its command line parameters and executes the string by invoking the shell on it, a devious Alice will supply a command line parameter containing, besides some dummy initial parameter, a statement-terminating semicolon and then a new command such as `/bin/sh`, with the result that the (root) shell invoked by `system` will first execute `/usr/bin/cat` with the dummy parameter and then a second command of `/bin/sh`, which will give a root shell to Alice. This is another example of privilege escalation through an unsanitised command line parameter. To defeat this attack, the programmer should avoid `system()` and instead use a function from the `execv()` family, where the command to be executed is clearly separated from its parameters[5].

Environment variables form a particularly insidious portion of the attack surface of the program because it is easy to overlook them as input. They are often not even mentioned in the documentation of the (setuid) program we are examining, because they are system-level facilities rather than features of the specific program: the `man` page for `chsh` will not mention `PATH`. Also, while `PATH` is pretty well known by most command-line users, other relevant environment variables are little-known outside the realm of C developers, for example the environment variables that control dynamic linking.

What is dynamic linking in the first place? Your program calls library functions. With static linking, both the code you wrote and the code of the library functions you call is linked into a static executable. If you write ten different programs that call the same library function, the

[5] If you are not already familiar with the mentioned POSIX utilities or system calls, you are encouraged to study their `man` pages, otherwise the understanding you might gain from reading this text will be only superficial.

code of that function is replicated 10 times into the statically compiled executables, taking up more space. Also, if one of the library functions is updated, for example to fix a bug (maybe even to patch a security vulnerability), then all your executables still contain the flawed version of the library function until they are recompiled and relinked against the new one. For this reason the default behaviour, if you don't tell the compiler to do otherwise, is instead to link libraries *dynamically*. Then you don't have multiple copies of the object code of the library functions and you always pick up the latest version installed on the system. What some people might not know is that there are environment variables that let you manipulate the behaviour of the dynamic linker: `LD_LIBRARY_PATH` is a bit like `PATH` but for searching dynamic libraries to be loaded, whereas `LD_PRELOAD` specifies a list of libraries that will be loaded first, whether the program requests them or not. These facilities are commonly used to supply an alternative version of a function that is instrumented for debugging, without having to recompile the executable to be debugged. Clearly this allows you to substitute arbitrary code instead of the code of the original program and for this reason `LD_LIBRARY_PATH` and `LD_PRELOAD` are ignored if the effective uid is not the same as the real uid, to avoid the obvious privilege escalation vulnerability. You will be doing an exercise, as part of your SEED lab, in which you will replace a library function with your own, and you'll check in which cases the substitution actually takes place and in which cases it doesn't.

However the feature interaction can be complex and, even though system programmers have thought about the potential for this facility to open up security holes, they have nonetheless been caught out. Here is a related incident. In OS X 10.10 "Yosemite", released in 2014, Apple introduced an environment variable called `DYLD_PRINT_TO_FILE` that allowed the developer to specify a file to which the dynamic linker `dyld` would output its debugging information. The baseline vulnerability is that this allowed a malicious actor to force a setuid-root program (`dyld`) to write to a protected system file such as `/etc/passwd`, although the attacker would not be able to choose what to write in it. So this could lead to denial of service (by corrupting crucial files) but it was hard to turn it into a privilege escalation exploit. Unfortunately the programmers of `dyld` also made another mistake: when `dyld` dropped its privileges, it did not close its open file descriptors, meaning that a careful attacker could still write to the open file under root privileges. Now *that* was quite disastrous and indeed it allowed the attacker to escalate privileges by writing into protected system files. Indeed, an exploit was found in the wild that modified `/etc/sudoers`, giving the attacker the right to

run anything as root without entering a password. One way of patching this vulnerability is to sanitise the environment variable before using it, so as to disable writing to system files (but you have to be extra careful because the attacker might have linked one of their files to a system file). A safer approach is, as we said earlier regarding `LD_LIBRARY_PATH` and `LD_PRELOAD`, not to honour the redirection at all when the real and effective uid are different.

We note in passing that, as some readers might have independently noted, the program featured in question 4 could also be exploited in another way. It accepts user input of unbounded size into a buffer of fixed size and it is therefore subject to a *buffer overflow* attack, which we shall study in chapter 4 on page 49.

The keenest among you might wish to consider what are the dangers of setuid scripts, as opposed to compiled programs, and why most modern versions of Unix ignore the setuid bit on scripts, defined as executables that start with `#!`.

3.1.3 Programs that spawn other programs

It is common for programs used and maintained by programmers to acquire, over time, additional functionality that makes the programmer's life more convenient. For example a text editor might start small but then acquire the ability to spawn a shell (`M-x shell` in Emacs). Of course this augments the attack surface for the program in ways that might not be obvious at first.

Always assume that any sufficiently large program might have, lurking somewhere, a little-known facility to spawn other programs, including perhaps a shell. Ask yourself if this facility could be triggered by supplying a malicious data file to a user of the program.

You should by now be familiar with LaTeX, a document processing system that takes a textual input and produces a neatly typeset output. Did you know it includes a command (namely `\write`) for writing to a file? Did you know that there is a special variation of that command (namely `\write18`) that invokes an external program? When I tell you the story this way, it should be obvious that this facility could be misused by someone who sent you some LaTeX source, which you might unsuspectingly compile under your own user id. This is why recent LaTeX distributions restrict that facility to spawn only programs from a whitelist[6].

But the general principle still applies. Whenever a complex program processes externally-supplied data, assume that the program (particu-

[6]See the relevant entry in the TeX FAQ for more details: https://texfaq.org/FAQ-spawnprog.

larly if it is itself programmable) might include an obscure facility to spawn other programs, and try to figure out if it could be triggered and exploited by sending the target program a suitably crafted malicious input.

Chapter 4

Buffer overflow

Textbook

Study Chapter 4 in W. Du, *Computer Security* 3rd ed.

SEED labs

Complete the following lab: Software Security | Buffer Overflow Attack Lab, Set-UID Version (tasks 3 and 4; all others optional).

The buffer overflow is a classic vulnerability that has been exploited in the wild since at least the 1980s. Sadly, to this day, programmers who should know better[1] continue to write code vulnerable to buffer overflows, and criminals continue to exploit such flaws.

The core idea is that the programmer sets aside a buffer meant to receive user-supplied data, and that the malicious user manages to write

[1]Part of the blame should be ascribed to the choice of programming language rather than to the competence of the programmer. A low-level language like C, which is still the tool of choice in many contexts owing to efficiency requirements and backwards compatibility with existing code bases, makes it very cumbersome to perform elementary operations safely. Try to write a C function that returns a string made of all the command line arguments of your program, concatenated with a space. This would be a one-liner in many higher level languages but it is a chore in C, and an invitation to hack up some quick-and-dirty solution that will suffer from buffer overflow. Of course a smart programmer will be careful and will be able to write safe code for that task; it's not rocket science, it's just a bit tedious and verbose. (Try it. At least once. I mean it.) But my viewpoint is that it is better to invest the precious attention and intelligence of the programmer in solving the novel difficult problems, rather than in avoiding the same decades-old pitfalls. That may be achieved by picking a more appropriate language for most parts of the project.

more data into it than the buffer was designed to contain. The additional data that spills over corrupts the adjacent area of memory, violating the model offered by the programming language abstraction. This is likely to cause some malfunction or crash. With some ingenuity, the attacker can arrange for the additional data to hijack the control flow of the original program and thus to execute some attacker-supplied code, which will run with the privileges of the attacked program.

Try the above-mentioned SEED lab first. If you can solve tasks 3 and 4 on your own already, you may skip ahead to Chapter 5 on page 69 of this handout and you may skip Chapter 4 of your W. Du, *Computer Security* 3rd ed textbook[2]. Otherwise continue here, do tasks 3 and 4 in the SEED lab to verify that you are able to apply what you understood, and consult the textbook if you get stuck.

4.1 Overwriting memory

The first ingredient of a buffer overflow vulnerability (the programmer's cardinal sin) is *accepting user input of unknown length without checking that it will fit* in the allocated buffer. Historically the C language has offered nasty library functions such as `gets()` and `strcpy()` that did just that, promoting and perpetuating this unsafe habit. Of course such functions are now deprecated[3]; but simply banning a few known-to-be-vulnerable-functions is not a solution, since there is no limit to the ways in which an ingenious fool of a programmer could write code that accepts unsanitised input without checking its length.

This first ingredient allows the malicious attacker to corrupt other data structures. For example, imagine a program on an online commerce website intended to send out a small reward to a customer, say either 5 or 10 £. The amount of the reward is held in the integer `reward` variable, while the bank account into which the reward is to be paid is held in the `payee` variable, the content of which the customer should supply. Imagine the `payee` variable is a 18-byte text buffer, intended to hold a string of the form `12-34-56 78901234`[4]. If the malicious customer

[2] At the time of writing, this particular chapter is currently freely downloadable from Professor Du's website as a sample: https://www.handsonsecurity.net/files/chapters/buffer_overflow.pdf.

[3] Have a look at their `man` pages, and try to compile and run programs that include them: nowadays, depending on the compiler you use and the options you invoke it with, you might get a safety warning even at runtime (which the user sees), not just at compile time (which only the developer sees).

[4] The traditional "sort code and account-number" format that identifies a bank account in the UK; 6 digits for the sort code, interspersed with 2 dashes; then 1 space, 8 digits for the account number and 1 terminating null to signal the end of the

supplies an overlong bank account number that spills over the end of the `payee` variable, maybe he can overwrite the `reward` variable as well. Try this program:

```
#include <stdio.h> /* printf(), gets() */
#include <stdlib.h> /* rand(), srand() */
#include <time.h> /* time() */

int main() {
  char name[20];
  int reward;
  char payee[18];

  srand(time(NULL)); /* seed the RNG */

  if (rand() % 2) { /* flip a coin (odd or even) to pick reward */
    reward = 5;
  } else {
    reward = 10;
  }
  printf("Your name? ");
  gets(name);
  printf("Your sort code and bank account? ");
  gets(payee);

  printf("Congratulations, %s! I'll send %d GBP to %s.\n",
     name, reward, payee);
  return 0;
}
```

Listing 4.1: buffer–overflow–1.c

Running this program and feeding it some typical input would result in a dialogue similar to this one:

```
Your name? Honest Joe
Your sort code and bank account? 12-12-12 34563456
Congratulations, Honest Joe! I'll send 10 GBP to 12-12-12
 34563456.
```

You'll notice that the program "flips a coin" and then issues a reward of either 5 or 10 pounds accordingly. If you are Sneaky Joe rather than Honest Joe, you might not be satisfied with such amounts: without changing the source code, can you try feeding the program different inputs in order to make it pay out a larger value?

string using the C convention. Total, 18 bytes.

We don't actually know where the program stores its name, reward and payee variables, nor in what order: at the abstraction layer of the C source code, the compiler could put them anywhere it likes, and in any order it likes. But let's make the hypothesis that they'll appear somewhere in memory, either in the same order as in the source code, or possibly in reverse order. If that's the case, since our target variable `reward` is sandwiched between two strings that we can input, whatever the compiler does we can hopefully spill into `reward` with either one or the other.

Let's first assume that the variables appear in memory in the same order as in the program listing: first `name`, then `reward`, then `payee`. Then, by supplying a customer name much longer than 20 characters, we ought to be able to overwrite `reward`, and possibly even `payee` (though we wouldn't notice the latter because we'd later overwrite those locations again with our own payee value). Let's try.

```
Your name? 01234567890123456789012345678901234567890123456789
Your sort code and bank account? 12-34-56 78901234
Congratulations, 01234567890123456789012345678901234567890123456789!
I'll send 10 GBP to 12-34-56 78901234.
Abort trap: 6
```

We entered a 50-character name. The amount is still 10 pounds (one of the two regular values) and the account number is the same legitimate-looking one that we entered. And then there is that mysterious "Abort trap: 6". What's up?

Let's leave aside that "Abort trap: 6" for the moment. Perhaps the compiler stored the `name` variable *after* the `reward` variable, so our over-long name had no chance to overwrite it. But then a long payee might. Let's try that.

```
Your name? Sneaky Joe
Your sort code and bank account? 01234567890123456789012345678901234567890123456789
Congratulations, 23456789! I'll send 10 GBP to
01234567890123456789012345678901234567890123456789.
```

Well, well... Now the `name` variable has clearly been overwritten, which tells us that, in the memory layout, `payee` comes before `name`. Also, this time we didn't get the "Abort trap" message. Strangely, the `reward` is still 10 pounds, one of the regular values, which suggests either a total fluke or that the `reward` variable appears in memory before both

of the two string variables. Hmmm. Surprising. Let's feed a different long payee to check if the 10 pounds had been a fluke[5].

```
Your name? Sneaky Joe again
Your sort code and bank account? ABCDEFGHIJKLMNOPQRSTUVWXYZabcdefghijklmno
Congratulations, ghijklmno! I'll send 10 GBP to
ABCDEFGHIJKLMNOPQRSTUVWXYZabcdefghijklmno.
```

OK, so we are still overwriting the name with the tail end of the payee, but the amount stays at 10 pounds (why never 5 pounds, by the way? but that's another question), suggesting that we are not touching the `reward` variable at all. So maybe our hypothesis that the compiler allocated the variables either in the same order as the listing, or in the reverse order, is not true: maybe the compiler is putting `reward` first, for some reason, followed by `payee` and finally `name`.

At this point in our investigation[6] we might as well cheat, and edit the source code to tell us what the addresses of these variables actually are. We insert three `printf` statements to show us the addresses of the three variables just after declaring them, and we leave the rest of the program as is.

[5] Though on reflection it can't be, because no subsequence in a string of ASCII digits would come out as 10 if interpreted as an integer.

[6] Which is at least in part intended to show you how you could go about figuring things out for yourself when you are not being guided by me or the textbook or a hacking tutorial you found on the net.

```c
#include <stdio.h> /* printf(), gets() */
#include <stdlib.h> /* rand(), srand() */
#include <time.h> /* time() */

int main() {
  char name[20];
  int reward;
  char payee[18];

  printf("&name: %p\n", &name);
  printf("&reward: %p\n", &reward);
  printf("&payee: %p\n", &payee);
  srand(time(NULL)); /* seed the RNG */

  if (rand() % 2) { /* flip a coin (even or odd) to pick reward */
    reward = 5;
  } else {
    reward = 10;
  }
  printf("Your name? ");
  gets(name);
  printf("Your sort code and bank account? ");
  gets(payee);

  printf("Congratulations, %s! I'll send %d GBP to %s.\n",
     name, reward, payee);
  return 0;
}
```

Listing 4.2: buffer–overflow–2.c

When we run this code, without even bothering to feed it any input we get...

```
&name:   0x7ff7bf312740
&reward: 0x7ff7bf312718
&payee:  0x7ff7bf312720
```

...which tells us that yes, first comes `reward`, allocated at address blabla...18h, then `payee`, at address blabla...20h, and finally `name`, at address blabla...40h, confirming the order we had logically deduced from our experiments. So there is no way that we are going to be able to overwrite `reward` with either of the other two inputs.

What's probably happening is that the compiler writers thought of buffer overflows and tried to make the programs a little safer against

them by putting the fixed-size variables first and the arrays last, so that the latter could not spill into the former. Of course this is not an absolute protection, if nothing else because we've just shown that we can overflow from one of the char array buffers into the other; but it did at least stop our attempt at overwriting the `reward`. If we turn off this particular countermeasure, the compiler will put the variables in memory according to a more "natural" order (perhaps the same as in our source code listing) and we may be able to perform our devious deeds. So let's recompile the same source code but with

```
gcc buffer-overflow-2.c -o buffer-overflow-2 -fno-stack-protector
```

(note the last command-line parameter supplied to the compiler). What we get when we run the newly-compiled executable is...

```
&name:   0x7ff7b437e760
&reward: 0x7ff7b437e75c
&payee:  0x7ff7b437e740
```

...where we see that the variables have now been allocated consecutively, and in reverse order compared to our declarations in the source code: `payee` at ...40h, `reward` at ...5ch and finally `name` at ...60h[7]. This means that, if we supply an overlong `payee`, we can overwrite the other two variables. Let's try:

```
Your name? Sneaky Joe
Your sort code and bank account? 012345678901234567890123456789012345678
Congratulations, 23456789! I'll send 825243960 GBP to
012345678901234567890123456789012345678.
```

Aha! Success! This time it worked, and instead of those measly 10 pounds we are now getting 825 million pounds. Of course the retailer is unlikely to even have that much money in her bank account, so this request is bound to be noticed and rejected and investigated. It would be wiser to be less greedy and request something that would not be picked up immediately as a serious anomaly, such as just 50 or 100 pounds instead of 10. Could we do that? Can you figure out what to supply as the payee string in order to cause the integer 50 to appear in the `reward` variable? You'll have to do a modest amount of arithmetic on the addresses to figure out exactly which byte of your payee string maps to which byte of the

[7] By the way: are they allocated consecutively, or are there any gaps between them? Can you do a bit of hex arithmetic to figure it out? And, if there are gaps, why did the compiler put them there?

integer variable, and then you might discover another problem related to null bytes... What is it? But, with some ingenuity, you might be able to make the reward exactly 50 pounds, and explain how you did it. So here is a little challenge for you: without changing the program, what input do you need to supply for the program to respond with "Congratulations, Sneaky Joe! I'll send 50 GBP to 11-22-33 44445555"? What problems did you encounter? Are there any further problems still lurking?

Anyway, the attack may not be perfect yet but we've demonstrated the main principle of buffer overflow. Provided that the programmer was sufficiently careless as to let you enter an overlong string into a buffer of limited size, and provided that other variables were allocated in memory at higher addresses than that of your buffer[8], then you are able to overwrite these other variables with your desired content.

4.2 Hijacking control flow

But that's *not at all* the full story: so far we have just overwritten data with other data. A buffer overflow can be much more devastating when it allows the attacker to run her own *code* on the target machine. Indeed most buffer overflow vulnerabilities that are exploited in the wild are used to run attacker code rather than merely overwrite data.

4.2.1 The call stack

How can this happen? You have to understand a bit about the call stack and the memory layout of a running program. Function calls can be nested to arbitrary depths and, when you return from a function, you resume execution at the point after the instruction that called the function. So, at the processor level, the machine code of the function cannot finish with the equivalent of a "go to" or "jump" instruction, because the place to go to could be different every time, depending on where the function had been called from. What happens instead is that the processor maintains a stack of addresses to return to and, when the calling code invokes a function, the processor pushes onto that stack the address of the instruction following the function call. On exit from the function, the processor pops the top item from the stack (which, if the stack has been kept balanced, is the return address the processor pushed before entering the function) and resumes execution at that address[9]. In machine code,

[8] And provided you don't run into problems with null bytes, newlines and the like...

[9] This mechanism is of fundamental importance to allow nested subroutines and is implemented natively by essentially every processor currently in existence. It was

that's all that happens: the "call function" operation is implemented as "push next address; then jump to function's address" and "return from function" is "pop the address on top of stack into the program counter", thus resuming execution from there. In higher level languages, what gets pushed on the stack is not just the return address but a rather more comprehensive "stack frame" including the local variables of the function (because each invocation of a function has its own distinct local copy of the variables) and possibly other items such as the function's parameters (depending on the calling convention) and a pointer to the previous frame. We shall look at stack frames in greater detail in the next chapter.

The main thing to note, for the purpose of understanding how a buffer overflow could result in executing the attacker's code, is that the return address goes on the stack, and any buffers that were declared as local variables of the function also go on the stack, and therefore there is a chance for the attacker to overwrite the return address, provided it comes "later" (meaning at a higher address) than the buffer that can be overflown. If the attacker can rewrite this return address, then she can tell the processor to execute code somewhere else. If she can supply her own code as part of the data she is sending, and if she can figure out where exactly her code will appear in memory during execution, and if she can figure out where exactly the return address to be rewritten will be, relative to the buffer into which she is writing, *then* she will be able to make the execution of the program continue into her own machine code, instead of returning to the code that called the function containing the vulnerable buffer.

4.2.2 The attacker's challenge

As you can see, there are a lot of conditions for the attack to work, and meeting all of them is challenging. In order to appreciate what is involved, and to understand the possible countermeasures against buffer overflow and their effectiveness (or lack thereof), you first need to experience the same difficulties as the attacker. Let's try attacking the previous program, which we know has a buffer overflow vulnerability, with the aim of executing our own devious code. As for the payload, we won't be terribly devious to begin with: we'll just invoke the `/bin/date` program, which merely prints today's date. Of course, in a real scenario, an attacker who can invoke and run `/bin/date` could also run any other program—including a shell, which is effectively a licence to run as many

invented in 1951 by David Wheeler (1927–2004), the first person in the world to earn a PhD in computer science, and incidentally the PhD supervisor of the PhD supervisor of my PhD supervisor here at the Computer Lab in Cambridge.

other commands as you wish, and under the privileges of the vulnerable program. So a successful buffer overflow attack on, say, a setuid root program would allow the attacker to run any desired command as root and essentially own the machine.

The payload itself, that is to say the devious code that we wish to run, has to be written in machine code. We can't just write the string `/bin/date`: we must write the bytes of the machine code instructions of a code fragment that will run the `/bin/date` executable[10]. Instead, we need to know how the operating system wants us to call an external program from machine code[11], but the knowledge about crafting the payload in assembly language is rather independent of that of creating the attack vector that exploits the buffer overflow vulnerability (or any other for that matter) so we won't sidetrack into that for now. If you wish to learn how to write your shellcode, which is an optional detour not included in this course's syllabus, you may start with Section 4.7 of the W. Du, *Computer Security* 3rd ed textbook and practice with the corresponding Shellcode SEED lab.

Let's instead assume the payload is already available. Script kiddies would typically find a suitably destructive payload online anyway. Here, in the `payload` variable in listing 4.3, is some x86-32 machine code[12] that will invoke `/bin/date`. Our job is to find a way to execute it.

First, just to reassure ourselves that it works, let's straigthforwardly call this machine code from within a C program. We load the bytes into a character buffer, we cast the buffer as a function and we invoke the function. If we compile[13] and run our C program, it prints the address of our machine code payload[14] and then runs it, which causes the execution of `/bin/date`.

[10] We also can't simply write a C program that invokes `/bin/date`, compile it and use the bytes of the compiled object code.

[11] The details are of course OS-specific and architecture-specific. There is a special software interrupt that is used to invoke system calls; a certain register must be loaded with a number that identifies a particular system call, in our case `execve()` or something similar; other registers must be loaded with the parameters required by the system call, which in our case will certainly include a string with the path of the program to be executed, and potentially other strings representing its command line parameters.

[12] Your textbook, and the associated SEED website, has examples and exercises covering both 32-bit and 64-bit. It won't hurt to try both but, for the purpose of understanding how things work, it's fine to stick to 32-bit only.

[13] We must compile with, on Ubuntu Linux, `gcc run-bin-date.c -o run-bin-date -z execstack -m32`. The `-z execstack` option disables an anti-buffer-overflow protection that makes the stack non-executable, and the `-m32` option generates 32-bit code as opposed to 64-bit code, since our machine code payload uses 32-bit opcodes.

[14] On the stack, of course, since it's inside a local variable of the `main()` function.

```c
#include <stdio.h> /* printf() */

int main() {
  const char* payload =
    /* x86-32 machine code to run /bin/date */
    "\x31\xc0\x50\x68\x64\x61\x74\x65"
    "\x68\x62\x69\x6e\x2f\x68\x2f\x2f\x2f\x2f"
    "\x89\xe3\x50\x53\x89\xe1\x99\xb0\x0b\xcd\x80";
  void (*pf)() = (void(*)()) payload;
  printf("&payload: %p\n", &payload);
  pf();
  return 0;
}
```

Listing 4.3: run–bin–date.c

This gives us a glimpse of what we need to do to attack our program: we supply our payload as a black box and then we must *overwrite*[15] the return address on the stack frame with the start address of our payload. Once that's done, on return from the vulnerable main function, the code will run our payload—and, in this case, print the current date and time.

We note that our payload is 29 bytes, which is larger than either of the buffers in our target program `buffer-overflow-2.c` from Listing 4.2 on page 54. Do we care? It seems like we should, but then maybe not. Exercise for the reader: what should we do? Hint: the payload does not have to be at offset 0 within the data we put in the vulnerable buffer.

One of the problems for the attacker is that **the address of the buffer** to be overflown (from which we may deduce the address of our machine code payload in memory by adding the relevant offset) **is generally not known**. Here we cheat (we make the novice attacker's life easier) by having the vulnerable program print it out.

But how would we address this issue in general? This is a crucial problem that you will need to practice with in the SEED lab if you want to master it. The general idea is as follows. If we don't know the absolute address where our buffer will be loaded, then we also don't know the absolute address of our payload, so we can't overwrite the return address with that exact value. If, however, we can narrow down that unknown address to a sufficiently small range of addresses, we may build a *NOP sled* of that length before our payload such that, provided that the return address points within the sled, execution will eventually transfer to our

[15] We haven't done that yet—the `run-bin-date.c` program does nothing more than demonstrate that the machine code payload works, but it is not an attack per se.

payload. We discuss this idea later in this section. If you get stuck, the W. Du, *Computer Security* 3rd ed textbook has a detailed treatment.

The second key problem is that we don't know exactly what the stack frame of the vulnerable function of the target program looks like, and therefore (even if we knew the start address of the buffer) **we don't know** at what memory address (or, equivalently, **at what offset from the start of the buffer**) **we'll find the return address** that we need to overwrite. Here the general idea is that, if we can narrow down that location to a sufficiently small range of addresses, then we may overwrite *each* of those addresses with the desired destination, and one of them will have the desired effect. Again, you'll need to practice with the SEED lab in order to master this technique, known as *stack spraying*. We discuss it later in this section and there is a detailed treatment in your textbook.

From the pedagogical viewpoint of trying things out we also have a problem of a different kind: given the ubiquity of buffer overflow vulnerabilities, the good guys have introduced a variety of countermeasures into operating systems and C compilers that defeat the basic versions of this attack. We have already encountered a couple of them. Since our purpose at the moment is to understand how things work, we'll simply disable those protections, teleporting ourselves back to the 1980s when none of them existed, so that we may retrace the steps of the people who first discovered and exploited buffer overflows. Once we have gained sufficient proficiency we might turn them back on, one by one, and figure out whether and how they might be circumvented. The protections we are going to disable, together with the instructions for disabling them when compiling with gcc on Ubuntu Linux 20.04 LTS[16], are as follows.

Stack canary A mechanism that detects, with high probability, whether the return address has been overwritten, by adding a random marker before it that would also be overwritten, and by checking for its presence before dereferencing the return address. Disable it by adding the `-fno-stack-protector` option to `gcc`.

Non-executable stack A mechanism, supported by the processor's hardware, whereby the processor refuses to execute machine instructions if they reside on the stack segment rather than on the code segment. Disable it by adding the `-z execstack` option to `gcc`.

Address Space Layout Randomization (ASLR) A mechanism that ensures that the various segments of the program (in particular the

[16]Equivalent methods will generally exist for other compilers and other operating systems, of course, but it's worth being specific to allow you to reproduce these results more easily, especially while you are still learning.

stack) are not always loaded at the same absolute memory address at every execution. Disable it by typing `sudo sysctl -w kernel.randomize_va_space=0` at the terminal.

You may check the effect of ASLR by running `buffer-overflow-2` several times. When you do so with ASLR still on, the address of the buffer should change at every invocation, whereas with ASLR off it will stay the same. This suggests that, assuming ASLR is off, if you are able to figure out the address of the buffer in a pre-attack trial, you may then construct an input that exploits this knowledge. For example, if you have the source code of the program you are attacking, you might instrument it to print out the address of the buffer as we did in `buffer-overflow-2.c` (Listing 4.2 on page 54). Another option might be to run the program under a debugger, as demonstrated in your W. Du, *Computer Security* 3rd ed textbook in Section 4.5.3. Note that the address of a buffer on the stack may not be exactly identical with and without the debugger because the debugger itself may have pushed additional data on the stack before running the program. But you will still be able to get useful hints, and the relative distances of items within a given stack frame will not change—for example the distance between the return address and the start of the buffer will not vary, even though the absolute address of the start of the buffer almost certainly will.

4.2.3 Stack memory layout

To make sense of the guesswork that will be necessary for the attacker, it is useful to have a somewhat clearer understanding of the memory layout of the stack. A common source of confusion is the ambiguity in graphical diagrams between high and low addresses (it is logical and customary to put "high" addresses at the top of the diagram; but if instead you print a region of memory in a debugger then the printout will run from top to bottom, and therefore the higher addresses will be lower down the page), compounded by the fact that the stack usually grows downwards (hence the "top of the stack" is, confusingly, at the lowest memory address). The issue is then made even worse by the fact that memory is addressed by words, but the bytes within the words also have an order, and in x86 this is often little-endian[17], which is the reverse of how you usually write down a multi-digit numeral such as 142 (one hundred and forty two, where the most significant digit, namely 1, worth one hundred, is the one that comes first). The convention I shall use in this lecture course, unless

[17]Little-endian means putting the least significant digit first, or leftmost, or at the lowest memory address. The opposite is big-endian.

otherwise noted, shall be that in a vertical diagram the high addresses are at the top[18], and in a horizontal diagram they are on the right. If a memory word is shown as a horizontal slice within a vertical pile of words, the bytes within that word will have increasing addresses from left to right[19].

With that notational detail out of the way, the layout of a stack frame varies somewhat between the original[20] 32-bit x86 architecture (formally IA-32), and the modern 64-bit x86-64 (formally AMD64) architecture[21]. The most salient difference is that, whereas in x86-32 the function parameters are passed on the stack, in x86-64 they are passed in registers (since registers are plentiful there) unless there are 7 or more parameters. That's aside from the even more obvious difference, namely that words are 4 bytes in x86-32 and 8 bytes in x86-64. There are other details but we won't go into them.

The stack frame has a logical "base", and then (perhaps strangely) some data both above it (the return address and then the function parameters) and below it (the local variables of the function). A specialised processor register called the base pointer[22] (ebp in x86-32 and rbp in x86-64) points at this base, and the compiled code refers to the function parameters and the local variables as the base pointer (whose value varies at runtime) plus or minus a fixed offset (whose value is fixed at compile time). The word pointed to by the base pointer contains the "saved" or "previous" base pointer (the base pointer for the stack frame of the parent function that called this one), whereas the following word contains the return address. The words after the return address contain

[18] Thus as the name suggests, but the opposite of what you see in a debugger dump.

[19] And thus, if the word represents a little-endian numeral, its bytes will be reversed compared to the normal writing order: the number 3 in 32 bits, normally written as `00000000 00000000 00000000 00000011` in binary or `0x00000003` in hex, will consist of a byte of value 3 on the leftmost and least significant position, in the lowest address, followed by three bytes of value 0, as in `03h 00h 00h 00h`.

[20] And now somewhat obsolete, though still in use in some contexts.

[21] Of course x86 in 32 and 64 bit flavours are not the only processor architectures in the world, particularly now that Apple's ARM-based chips have made their way into mainstream laptops and desktops, but we focus specifically on the Intel-derived architecture for several reasons, including (a) a desire for the examples to be concrete and easily reproducible rather than general but theoretical; (b) a desire to make the course consistent with the recommended W. Du, *Computer Security* 3rd ed textbook; (c) a desire to let you gain practical experience through the existing SEED labs; (d) the fact that most of the deployed computers that you would end up having to attack or defend in real life are still x86-based. In any case, once you gain hands-on experience on a specific architecture, it's relatively easy to generalise the ideas to any other. You will have to do that throughout your career regardless of what I teach you this year anyway.

[22] Also sometimes "frame pointer".

the function parameters[23]. The words before the base pointer contain the local variables. As we have seen earlier in this section, the compiler is at liberty to decide the order in which the local variables appear in the stack frame, although the "vanilla" arrangement, barring optimisations and security safeguards, is to place the first local variable just below the base, the second one below that and so forth, which results in the variables appearing in memory in reverse order than in the source code listing (the last variable in the source code is at the lowest memory address). The "vanilla" arrangement for function parameters, instead, is to place the first one just above the return address, the second one after that and so forth, with the result that the parameters appear in natural order in memory rather than reversed. It should also be noted that there may be gaps between the variables, often to align the start of a variable with a word boundary but sometimes possibly also for other reasons.

The stack grows downwards, as mentioned: when function `a()` calls function `b()`, the stack frame for `b()` will appear at a lower memory address than the one for `a()`.

With these preliminaries out of the way, our job is to craft an input that will cause our target program `buffer-overflow-2` (Listing 4.2 on page 54) to execute our binary payload. For demonstration purposes we shall use as payload the sequence of 29 machine code bytes that featured in `run-bin-date.c` (Listing 4.3 on page 59) and which merely prints the current date; but of course, once the attack vector has been developed, the attacker could deliver any arbitrary payload with it.

We must generate a sequence of bytes that will be read by line 23 (the `gets(payee)` instruction) in Listing 4.2 on page 54. This gives us the additional constraint[24] that our sequence may not contain the newline character, because that would terminate the input for `gets()`. Let's call B (for Base of Buffer) the absolute memory address of the vulnerable buffer at runtime, in other words `&payee`; let's call R the absolute memory address of the place in the stack frame where the return address is stored; and let's call P the absolute memory address of our payload[25]. (It may be a good idea for you to draw diagrams as we go

[23] Except that, as we said, in x86-64 the first 6 parameters are passed in registers, which saves both time and space.

[24] And will your sequence be allowed to contain any null bytes? Why, or why not? If yes, how would you enter them? More generally, how would you enter any bytes corresponding to characters not on your keyboard?

[25] It might perhaps make more sense to speak first of the offset oP of our payload within our sequence of bytes, with $P = B + oP$. We have control over oP, since we can put our payload wherever we like within our attack sequence, and P is a derived quantity. But I found it clearer first to introduce single capital letters for the absolute addresses, and then digraphs starting with "o" for the offsets.

along, to make sure you internalise what we are talking about.) What we need to do, having loaded our payload at offset oP within our attack sequence, is to figure out the offset oR of the return address ($oR = R - B$) and then poke the value P at that offset in our sequence, making sure it won't overlap with the payload itself.

If we are able to run the vulnerable program under a debugger (see Section 4.5.3 of the W. Du, *Computer Security* 3rd ed textbook), we may discover oR by setting a breakpoint inside our function and then inspecting the value of the base pointer (which is one word less than the return address, hence $R - 4$ or $R - 8$ for x86-32 or x86-64 respectively) and the address B of the buffer. Even though R and B won't be the same as when running without the debugger, since the whole stack frame may have shifted down, their difference oR will be.

What if we are not able to run the target binary under a debugger, perhaps because it is running on a remote server[26]? We'll have to guess. After all, we more or less know what the stack frame contains (aside from possible unexpected gaps) so there aren't that many places where the return address could be, relative to B. We might as well poke our redirected address P in all of those places, and then one of these guesses will be correct and will cause our payload to be executed.

But what about B? In this toy example we are safe, because the vulnerable program itself helpfully prints out B. However this does not usually happen. So what should we do if we have to guess both B and R? If we don't know B, we can't derive P either, which means the previous technique of writing P in many places in our sequence won't work, because we are not sure what P we should be using.

Well, let's pretend we can venture a guess that B will be within a certain limited range of possible addresses. After all, without ASLR, the stack always starts in the same place, and the typical non-recursive program doesn't have a terribly deep stack in terms of number of stack frames. A useful technique is the so-called "NOP sled", where NOP is the opcode for the "no-operation" machine code instruction—a one-byte "space filler" instruction (opcode 0x90 on x86) that does nothing and moves on to the next instruction. If we prefix our payload with a long sequence of NOPs, say n NOPs, then not only P but any address between $P - n$ and P will be a valid starting point for our payload! Because if you jump anywhere in the sled of NOPs, you will simply slide right from NOP to NOP until you reach the start of our payload.

Provided you can arrange for the payload, the NOP sled and the region sprayed with return addresses not to overlap with each other, you may craft an attack sequence that will work first time even if you are

[26] Cfr. SEED lab "Software security | Buffer overflow attack lab (Server version)".

not sure about B and oR. See Section 4.6 of the W. Du, *Computer Security* 3rd ed textbook and practice attacking `buffer-overflow-1.c` which doesn't print its B.

I'll say it again: to pass this course, it is very important for you not just to understand these ideas superficially. You must acquire the hands-on experience of having been able to recreate an exploit against a vulnerable program, computing all the required values and seeing the attack succeed—in the same way that you cannot learn programming without writing and debugging your own programs. I highly recommend completing the SEED labs indicated throughout the text, even though they will not be assessed and they do not attract course credits. As a reward, I shall do my best to write exam questions that will greatly favour candidates who have gained the experience one acquires through completing the SEED labs.

4.3 Countermeasures and counter-countermeasures

Buffer overflows have been exploited in the wild for decades and it is somewhat embarrassing to the software engineering profession that this is still happening. A variety of countermeasures have been devised and some are commonly adopted by default, as you noticed when we had to disable three of them to do the SEED lab. The ones we disabled were first mentioned on page 60.

- The **stack canary**[27] incurs a little space and time overhead (usually acceptable, but there are trade-offs) to generate and store a marker on creation of the stack frame and to check it on exit. The more secure implementations use a random marker, whereas those that use a static marker are obviously much easier to defeat. However, even with a random marker, the code that performs the check must compare the marker with a reference value stored somewhere when the marker was assigned, so it is not entirely inconceivable that the attack code might fish it out and reproduce it into the intended place before the check is performed.

- The **non-executable stack**[28] ensures that machine code residing in the stack segment will not run. Most modern CPUs usually offer an NX (non-executable) bit in hardware, through which this countermeasure may be offered with negligible overhead. Attack

[27] From the practice of taking a caged canary when visiting a coal mine as a detector of poisonous gases.

[28] Also referred to as DEP on M$ Windows, for "Data Execution Prevention".

code might however store the payload elsewhere (e.g. on the heap rather than the stack) or use return-to-libc (cfr. Chapter 5 on page 69) or return-oriented programming to bypass this protection.

- **Address Space Layout Randomisation (ASLR)** is an operating-system-level countermeasure whereby the various segments of a process (code, stack, heap, libraries) are allocated in memory at random addresses rather than in predictable locations. The total number of bits of randomness in the placement of the segments determines the effort required for a brute-force search of the correct values by an attacker. Small search spaces (say 16 bits, hence 2^{16} variations) can be brute-forced in minutes. To make brute-force attacks harder, the OS may enforce a delay before launching an executable that has already crashed a certain number of times. But attackers may be able to obtain the ASLR-protected addresses through side-channel leaks from the memory-management unit.

- The root cause of buffer overflows is writing more data into a buffer than will fit into it. Nowadays, **programming languages enforcing mandatory bounds checking** on every array access, such as Python and Java, consider this overhead as a reasonable price to pay to rule out buffer overflows[29]. This is a great choice when the developer has the luxury of starting from scratch in a new language, but it may not help when the task is to secure an existing codebase, or when the programming task is sufficiently low-level that it calls for C, C++, assembly or some other language without bounds checking.

- As a glimpse into the future, a radical new approach to eliminate buffer overflows and other memory safety violations is **CHERI**[30] (Capability Hardware Enhanced RISC Instructions), an ongoing research project originated by Robert Watson and led by him with Simon Moore and Peter Sewell here at Cambridge and in collaboration with Peter Neumann at SRI International. CHERI rethinks software security from the ground up, from the processor hardware to the compiler toolchain to the system software. The core idea here is *capabilities*, implemented as pointers augmented with validity bounds and other security metadata. The security metadata is unconditionally checked by the hardware at every access, with negligible runtime overheads. This approach allows the system to run a

[29] Of course this does not protect against buffer overflows in the interpreter itself, but that is a separate issue and in any case the potentially vulnerable code base is now limited to the interpreter rather than all the possible programs that it could run.

[30] https://www.cl.cam.ac.uk/research/security/ctsrd/cheri/

Unix-like OS, CheriBSD, and to offer memory safety to C and C++ programs, mitigating a large fraction of the major vulnerabilities that have affected other current systems. The purposeful co-design of hardware, software and semantics allows formal modelling of the architecture and the development of more extensive security proofs than would be possible in mainstream systems. The project has attracted many collaborators and well over a hundred million dollars of government and industry funding. Arm has built a hardware prototype, called Morello, where the CHERI architecture is built into an Arm processor. The Morello demonstrator board, which first shipped in January 2022, has initially been distributed to academic and industrial partners for experimentation.

In conclusion, buffer overflow is still one of the most frequently exploited vulnerabilities. As a future software professional you should turn on as many mitigations as are available, program defensively and with awareness of the problem, adopt languages with bounds checking wherever possible, and then probe[31] the systems in your organisation for any residual traces of buffer overflow. The arms race is still ongoing, there are still plenty of legacy systems, there are still plenty of uneducated programmers and a belt-and-braces strategy is necessary against such a frequently reoccurring vulnerability.

[31] Always with the prior consent of the people in charge! Don't get yourself into worse trouble while naïvely thinking you're doing good.

Chapter 5

Return to libc

Textbook

Study Chapter 5 in W. Du, *Computer Security* 3rd ed.

SEED labs

Complete the following lab: Software Security | Return-to-Libc Attack Lab 32-bit (tasks 1, 2, 3; others optional).

As we mentioned in Section 4.3 on page 65, one of the countermeasures against buffer overflow attacks is to mark the stack as not executable[1]. That way, even if the attacker manages to overflow a buffer on the stack and overwrite the return address of the function that accepted her overlong input, she will not be able to execute the malicious code in her payload. One counter-countermeasure to that is to smash the stack as before, overwriting the return address, but then to execute code that resides in a genuine code segment. The attacker cannot supply that code as part of the payload, since the attacker-supplied input is still in the overflown buffer on the stack, but she can refer to other code that is

[1]The first processor to allow this was the Intel 80286, introduced in 1982. It introduced *types* to the already-existing segments (Code segment, Stack segment, Data segment and Extra segment) defined by the 8086. These types mandated that instructions could only be fetched from the Code segment. But the operating systems of the time did not use this facility for security. The modern "no-execute" (NX) bit was introduced by the AMD Athlon 64 (2003) and it worked at the page level (finer granularity than segment level). This was the mechanism that was widely adopted as a buffer overflow mitigation in modern OSes.

already present in the memory accessible to the process under attack. In particular, the process will have typically been linked with the standard C library and therefore it will be possible to invoke libc system calls, including very general ones such as `system()` or `execve()` that will let the attacker run arbitrary executables already present on the system, including a shell.

Try the above-mentioned SEED lab first. If you can solve task 3 on your own already, you may skip ahead to Chapter 6 on page 81 and you may skip Chapter 5 of W. Du, *Computer Security* 3rd ed. Otherwise continue here, do tasks 1–3 in the SEED lab (4 and 5 optional) and consult the W. Du, *Computer Security* 3rd ed textbook and/or Section 4 of the SEED lab's briefing document if you get stuck.

The challenges for the attacker include:

- Finding the address of the desired system call in the target process's address space (easy).
- Passing the intended parameters in the way the system call expects them.
- Optionally (for stealth) cleaning up afterwards so that the process won't crash after exiting from the system call. If it crashed, this might raise red flags and make it easier for defenders to notice the attack.

None of the above is particularly difficult per se but, as usual, debugging a non-working attack is not easy: you need to get all the details just right and it is hard to get clues as to what might not be working as intended. I highly recommend practicing with the relevant SEED lab until you are able to get the attack to work. If you get stuck, keep a cool head and consider starting again from first principles, from a clean copy or snapshot of the SEED VM and with a fully updated guest OS, rather than from an image in which you had already done several other SEED labs. Do not give up. Try other approaches. If reasoning does not work, try brute force and then work backwards. One of the main difficulties is going to be the depth of the various rabbit-holes you might end up having to explore, including assembly language syntax, function call conventions, debugger syntax and so forth. You are not expected to know everything, but you are expected to learn how to learn the bits of information that are required for you to complete the task at hand—this is a crucial skill for a professional computer scientist that will serve you in many more contexts than just implementing this return-to-libc attack. The relevant chapter in the W. Du, *Computer Security* 3rd ed textbook is very detailed and I recommend referring to it if you run into trouble.

The online briefing document of this SEED lab has itself a useful Section 4 ("Guidelines: Understanding the Function Call Mechanism") that is well worth studying.

The main prerequisite to carry out this attack is a sufficiently thorough low-level understanding of what happens on the stack when the flow of control enters and exits a function. Of course you are aware that a stack frame is created when entering the function and is dismantled on exit, but we need to be rather more specific. The details are fairly platform-specific and will be different for systems other than 32-bit Intel executables but it is much more instructive to get things to work for a specific instance than to handwave about generalities. Once you got things going for one particular case, adapting to other platforms will be relatively easy. For concreteness, we stick to x86-32 in this course.

5.1 Function calling conventions and stack frames

In machine code, transferring control to another portion of the code is simply a matter of executing a jump instruction that reloads the instruction pointer with a new address. The difference between that and a subroutine call is that, on completion of the subroutine, we want execution to resume at the instruction following the one that called the subroutine; and so, given that the subroutine may be called from multiple places, we cannot put a jump at the end of it because the place to jump back to depends on where the subroutine was called from. So the "call subroutine" instruction in machine code[2] consists of pushing the instruction pointer on the stack and then executing a jump to the subroutine, whereas returning from the subroutine consists of popping back the top address on the stack into the instruction pointer, and continuing execution from there. The processor knows nothing about subroutine parameters. Each subroutine author is, in theory, free to come up with her own conventions about (say) passing parameters through registers, or through designated memory locations, or on the stack (provided it is kept balanced) or whatever. When compiling higher level languages, however, it is necessary to stick to some common convention for interoperability. The x86-32[3] calling convention is as follows. Note that, in the following diagrams (courtesy of Wenliang Du), as per our standard convention in this course, higher memory addresses are higher up in the picture, and the stack grows downwards; so the logical "top of the stack",

[2] Mnemonic `CALL` in x86 and many other assembly languages.
[3] Also known as IA-32.

pointed to by the `esp` stack pointer processor register, is at the bottom of the diagram.

Since x86-32 has relatively few registers, the input parameters of the routine are passed on the stack[4]. The caller pushes the parameters on the stack one by one in reverse order[5] and then calls the routine. Calling the routine implicitly pushes the return address[6] on the stack after[7] the parameters.

The callee, on entry into the routine, pushes the `ebp` base pointer register on the stack (to save the old value for later) so as to be able to overwrite that register with the base of the current stack frame. The callee will restore the old `ebp` on exit, so that `ebp` always points at the frame for the current routine. Just after pushing `ebp` on the stack[8], the callee copies the stack pointer `esp` into `ebp`, thus making `ebp` point at the word on the stack containing the saved previous copy of `ebp`[9].

Immediately after that, the callee makes space for its local variables by subtracting an appropriate constant (determined at compile time) from the stack pointer `esp`. The space on the stack between the addresses pointed to by `esp` before and after the subtraction is where the local variables are stored. Each local variable is stored at a constant negative offset from the base pointer[10], whereas each input parameter is stored at a constant positive offset from the frame pointer.

Together, the instructions[11] that are always executed when entering

[4] At some performance cost; which is why architectures with an abundance of registers, including ARM and x86-64, pass parameters in registers where possible.

[5] Hence the first parameter (the last one pushed and thus, since the stack grows downwards, the one at the lowest address) is the one at the smallest positive offset from the base of the frame pointed to by `ebp`, and the routine's parameters are found in order when progressing upwards through memory from there.

[6] The return address is the crucial word that the attacker wishes to overwrite in order to hijack control flow—this continues to be so even in this situation of non-executable stack.

[7] On top of them on the stack, hence at a lower memory address.

[8] Which causes the stack pointer `esp` to move down, to point at the newly inserted word which is now the top-of-stack item.

[9] Note the initially confusing fact that the base pointer points "in the middle" of the stack frame rather than at the start of (= lowest address in) the stack frame. The base pointer points, as we said, at the word containing the saved previous base pointer. The word above that is the return address and the words above that are the input parameters of the routine. Below the saved base pointer are instead the local variables of the routine.

[10] Why do we not instead refer to local variables with positive offsets from the stack pointer? Because the routine could push other data onto the stack, which would change the stack pointer while still inside the routine. By referring to offsets from the base pointer, these remain constant for the lifetime of the local variables.

[11] The assembly language listings in this chapter, like the ones in your W. Du, *Computer Security* 3rd ed textbook, use the AT&T syntax, as is common for x86

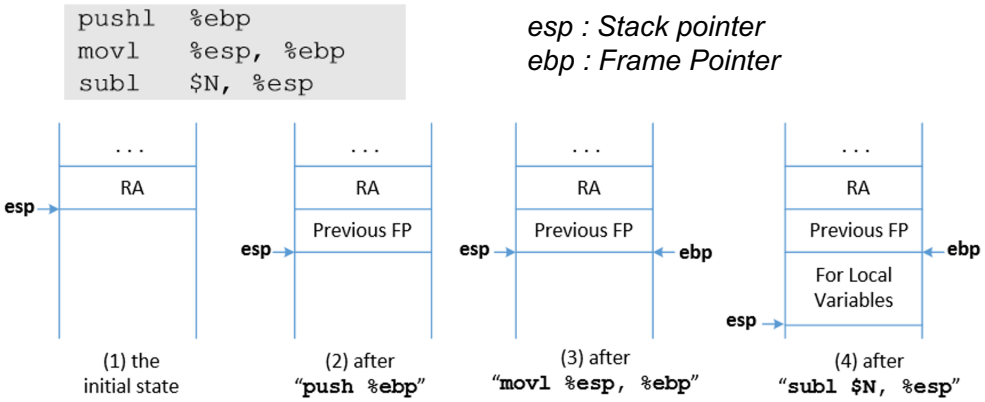


Figure 5.1: Function prologue *(courtesy Wenliang Du)*. NB: "Frame pointer", or FP, is the same thing as "base pointer", register `ebp`. RA stands for "Return Address".

the routine (the so-called *function prologue*) are:

```
1 # function prologue, executed on entering the callee
2 pushl %ebp # save the previous base pointer on the stack
3 movl %esp, %ebp # make %ebp point to it
4 subl $N, %esp # make space for callee's local vars
5 # end of function prologue
```

Listing 5.1: `prologue.asm`

This creates the bottom part of the stack frame for the function, the top part being the input parameters and the return address which were set up by the caller.

On exiting the function—or more precisely *just before* returning from the function—the *function epilogue* is executed, which undoes what the prologue did:

Unix systems. In AT&T syntax, source comes before destination and `movl A, B` means that A gets copied into B. There is also an alternative Intel syntax in which, confusingly, the order of parameters to a machine code instruction is reversed. AT&T syntax says "copy the source into the destination", whereas Intel syntax says "copy, into the destination, the source".

```
# function epilogue, executed on exiting the callee
movl %ebp, %esp # release the space of the local vars
popl %ebp # restore the previous base pointer
# end of function epilogue

ret # pop return address into instr pointer
```

Listing 5.2: `epilogue.asm`

The x86-32 instruction set has two instructions[12], `enter` and `leave`, that perform the prologue and epilogue respectively, so you may see them instead in assembly language listings. We show the expanded lower-level instructions for clarity about what is actually happening.

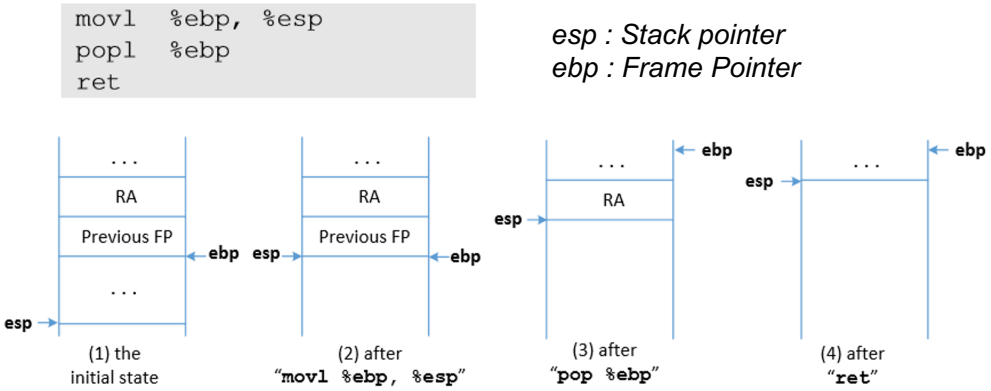


Figure 5.2: Function epilogue *(courtesy Wenliang Du)*.

5.2 Return-to-libc attack

Having established these preliminaries, we construct a return-to-libc attack to exploit a buffer overflow in presence of a non-executable-stack countermeasure. The core of the vulnerable program that you'll be attacking in the SEED lab is shown in Listing 5.3 on page 76 (slightly simplified from the actual program by removal of `#include`s and of code that computes and prints out some helpful hints). The vulnerable program is configured as setuid root and the objective of the attacker is to

[12]When the 8086 was introduced in 1978, they used to be macros provided by the assembler. With the 80186, in 1982, they were added as actual instructions of the processor. However, with the 80386 (1985), many compilers stopped using `enter` and went back to generating the sequence of `push`/`mov`/`sub` because it was easier to optimise and pipeline.

escalate privilege and spawn a root shell. Note that you won't need to include a machine code payload in your `badfile` to do that, since you are not allowed to execute any of the bytes you put on the stack anyway.

The `main()` function of the program accepts user input from a `badfile` (lines 11–12), whose first 1000 bytes are copied into the suitably-dimensioned `input` buffer (line 9). The `main()` function then calls the vulnerable `bof()` function, which copies the supplied string into a possibly under-dimensioned `buffer` (line 4, with the vulnerability we'll exploit). Then `bof()` returns, and finally so does `main()`.

The job of the attacker is:

1. Overflow the `buffer` declared in line 2 to overwrite the return address of `bof()` with the address for the `system()` system call. This in turn involves:
   (a) Finding the offset of the return address with respect to the start of the `buffer`. We already did this in Section 4.2.3 near page 64, so we won't dwell on how to do that: for simplicity, we assume we are able to run the program under a debugger to find this offset. Our target program in the SEED lab even helpfully prints out some of its internal addresses to make our job easier. In real life we would redo the things we did in Section 4.2.3, but here we concentrate on the new aspects of this particular attack.
   (b) Finding the address of `system()` in the target process's memory space. We assume that we can just look up the address by running the target program under a debugger (say `gdb`), once libc has been loaded[13]. We also assume that ASLR is off and that therefore we may reuse that result after exiting `gdb`.
2. Supply the correct parameters to `system()` to make it launch `/bin/sh`[14]. The `system()` function (check `man 3 system`) takes a string containing a command and runs it by forking a child process that runs `/bin/sh -c` with the supplied string as a one-shot command[15]. It

[13] In `gdb`, you may print the address of a library function such as `fopen` by typing `print &fopen`, provided that the function name symbol is known in the then-current context. For that, compile the program with the `-g` option to include symbolic debugging information, and run the program (to some breakpoint) in `gdb` before issuing the above command.

[14] The SEED lab has details of a countermeasure implemented by `/bin/dash` and `/bin/bash` whereby these shells drop privileges if they are executed under setuid. The lab thus suggests symlinking `/bin/sh` to `/bin/zsh` which does not have that countermeasure. It then shows you how to defeat that countermeasure in a later task.

[15] This means that, when we ask `system()` to run `/bin/sh`, there will in fact be two layers of `/bin/sh` nested inside each other!

```
1  int bof(char *str) {
2      char buffer[BUF_SIZE];
3      /**** printing of hints omitted for brevity ***/
4      strcpy(buffer, str); /* buffer overflow vulnerability */
5      return 1;
6  }
7
8  int main(int argc, char **argv) {
9     char input[1000];
10    FILE *badfile;
11    badfile = fopen("badfile", "r");
12    int length = fread(input, sizeof(char), 1000, badfile);
13    bof(input);
14    return 1;
15 }
```

Listing 5.3: retlib.c

takes one argument (a pointer to the string with the command) which we must supply on the stack in the usual way. To do that, in turn, we must:

(a) Arrange for the null-terminated C string `"/bin/sh"` to appear in the memory of the process. We might think of putting it at the start of our `badfile` but this won't work because the terminating null will cause `strcpy()` in line 4 to stop copying, meaning we won't overflow the `buffer` any more. The W. Du, *Computer Security* 3rd ed textbook and the SEED lab show you another way of making the string accessible to the target program, namely by declaring an environment variable[16].

(b) Figure out the memory address of the `"/bin/sh"` string. This of course depends on whether you placed it in an environment variable, in the `badfile`, or anywhere else; but in general it shouldn't be too hard. The SEED lab and the textbook show you how to do it for the environment variable, basically by writing a similar program and have it print out the address from within it. If instead you put it in the `badfile`, you'll have to figure out the base address of the buffer in step 1a

[16]It is of course also possible to insert the string in the `badfile` by putting it at the end, *after* the section that overwrites the return address: that's what I did when I tried the lab for myself. If you enjoy the hacking we are doing, try this alternative as well. The optional Task 4 in the SEED lab uses a similar approach for another purpose.

anyway; and, once you have that, you just add the offset that the string has in `badfile` to find the string's absolute address in memory.

(c) Poke the absolute memory address of that string in the correct position in the stack where `system()` will be looking for its first and only input parameter. This requires a little thought, and we'll look at it next.

3. Ensure that the program will exit gracefully, without crashing, once the execution of `system()` has completed. This is not a true necessity, in that the attack will work without it, but it would be nice to have it for covertness. To do that we must:

   (a) Figure out the address of the `exit()` C library function, which we do in the same way as we did for `system()` in step 1b—simply by looking it up in the debugger.

   (b) Figure out where on the stack the `system()` function will look for its return address, and poke the address of `exit` in there. This requires an investigation similar to the one for step 2c. We'll discuss both of these next.

Consider what happens during the execution of the vulnerable function `bof()` (cfr. Listing 5.3 on page 76). Until line 3, the memory layout is as in Figure 5.3 (a), which depicts all and only the stack frame for `bof`: `ebp` points at the word containing the previous `ebp`. Above that is the return address from `bof`, and above that the lone input parameter to `bof`, namely `str`. Below the word pointed to by `ebp` is the only local variable of `bof`, namely `buffer`. During the execution of the `strcpy()` function on line 4, your buffer overflow attack will overwrite various locations: the saved `main()`'s `ebp` with some irrelevant garbage of your invention; the Return Address from `bof()` with the address of `system()` (this being the crux of the attack); and then possibly other subsequent locations with other stuff. (What should that other stuff be?)

When we return from `bof()`, the epilogue gets executed: `ebp` is copied into `esp`, thus releasing the space of `bof()`'s local variables, and the top of stack (containing irrelevant garbage written by us) gets popped into `ebp`. This moves `esp` one further word up, to point at the location marked "Return Address" in Figure 5.3 (a), which our attack has already overwritten with the address of `system()` by now. Then the `ret` instruction is executed, which pops the address of `system()` into the instruction pointer, making the control flow jump there and bumping `esp` up by one more word. At this point we are in the situation depicted in Figure 5.3 (b).

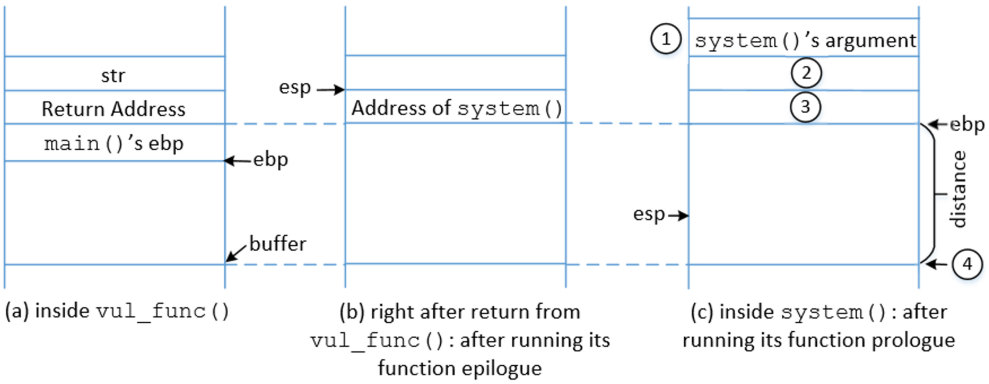


Figure 5.3: Memory map to understand the location of `system()`'s argument *(courtesy Wenliang Du)*.

Having entered `system()` (although without the regular process of pushing its parameters on the stack and then invoking `call`, as a regular caller would have done), the function prologue for `system()` is executed: `ebp` (now garbage) gets pushed on the stack, overwriting the address of `system()` in the slot marked ③, lowering the stack pointer to point at (= just below) that slot. The `esp` gets copied to the `ebp`, establishing the base pointer for the current function; some space is allocated for the local variables of `system()` by subtracting some amount from `esp` (but we don't care). And the situation is as depicted in Figure 5.3 (c).

Now, what's of interest to us is where will `system()` look for its stuff. It will assume that the location pointed by `ebp`, namely ③ in Figure 5.3 (c), contains the previous base pointer (not true but fine; we don't care); that the word above that, namely ②, is the return address from itself (and we *do* care about that: it's where we want to write the address of `exit()`); and finally that the word above that, at position ①, is the input argument it expects, that is to say the address of the string with the command to be executed (and we do care about this one as well: it's where we want to poke the address of `/bin/sh`).

The last piece of the puzzle for us is to figure out the absolute memory addresses of ①, ② and ③. We know ④, which is the base of the buffer: in step 1a on page 75 we assumed we'd be able to find that with previously known techniques, and indeed the SEED lab target program even just prints it out for us, to save us the trouble. Do we know the distance between ③ and ④? By looking at the diagram, yes, because ③ is the same as the location of the return address of `bof()`, which we obviously had to compute in step 1a in order to overwrite it in the first place. Since ③, ② and ① are consecutive adjacent words in memory, with that we

now have the absolute addresses of all of them. We now thus have all the information we need to craft our `badfile` and get a root shell out of the target program.

If you have the luxury of attacking a program that is totally under your control and that cannot "phone home", you might of course also find the right addresses by trial and error, without attempting to predict everything in advance to the last byte. But it is instructive to understand, at least once, why things are exactly where they are.

Note also the more advanced technique of **return-oriented programming** (ROP), in which you don't merely call a function already present in `libc` but instead you create your own code by chaining "gadgets", which are snippets of machine code instructions, terminated by `ret`, that perform elementary operations such as popping a value into a designated register. A chain of carefully chosen ROP gadgets has vast expressive power. If enough gadgets are available in the loaded libraries (and tools have been developed to search for them), arbitrary code may be built out of them.

Chapter 6

SQL injection

Textbook

Study Chapter 14 in W. Du, *Computer Security* 3rd ed.

SEED labs

Complete the following lab: Web Security | SQL Injection Attack Lab (tasks 2 and 3; other tasks optional).

SQL injection is another one of those old and well-known vulnerabilities that do not seem likely to go away soon, even though a reasonable and effective countermeasure is available.

There are many contexts in which data is stored in a database and some interface is provided for users to query or update the database directly from a web page or other kind of remote application. The idea is that the developer prepares a parametric SQL statement (along the lines of "show me all the rows with this field equal to X") and asks the user to supply the values for the parametric fields. The trouble is that the statement is constructed by pasting the user-supplied values into a template string and then feeding that string to the database engine, for it to be interpreted as an SQL statement. The core of the vulnerability is that the user input gets inserted into the template "as is" and then the result gets parsed as a command. Therefore, if the user input is not properly quoted, it may escape the boundaries of the input field and escalate from merely being data to becoming a command itself: the attacker has thus injected some SQL (which she was not supposed to be

able to write) into the system. The attacker is now able to read other data or even modify the database.

Try the above-mentioned SEED lab first. If you are able to complete task 3.3 on your own, you may skip the rest of this section and the corresponding book chapter, and move straight ahead to Chapter 7 on page 89. Otherwise, read on and do the SEED lab: task 1 is a trivial SQL one-liner with no security content and you may treat task 4 as optional, but do complete all of the six subtasks in 2 and 3. As usual, the textbook has further details if you get stuck.

Extra things you will have to get to grips with for this lab include Docker and PHP, as well as of course SQL, to which you were already exposed during the Databases course last year. As ever, the more you do the better for you[1], but time available is finite so do not feel you have to become an expert at any of these auxiliary topics in order to proceed. Figure out as much as you need to get going, and stay focused on the core business of SQL injection, which is not difficult. The online briefing document for the SEED lab contains as much of a basic primer on them as you need to get going, and there's always the W. Du, *Computer Security* 3rd ed textbook (or a web search) for more.

6.1 Quoting user input

Failing to quote user input correctly is the root cause of a great deal of security problems. It is a problem that resurfaces in many contexts: I would argue that not only SQL injection but also buffer overflow, cross-site scripting and all the cases where a program is abused for having invoked `system()` instead of a system call from the `execve()` family are also instances of that.

Most programmers are rubbish at quoting

Programmer incompetence at quoting user input is an inexhaustible source of vulnerabilities

Have a look at the classic "Little Bobby Tables" XKCD cartoon from 2007-10-10 which illustrates the SQL Injection attack concisely. What do

[1] Docker, in particular, is an interesting and versatile technology that is worth learning more about, even aside from this course; and you'll amortise the effort because we'll use it in a few other SEED labs. PHP, on the other hand, is not something I'd recommend investing a lot of mental energies into, unless you end up having to maintain other people's websites.

you think the vulnerable SQL statement in the school's database might look like, and how exactly does the hacker mum's devious input break it?

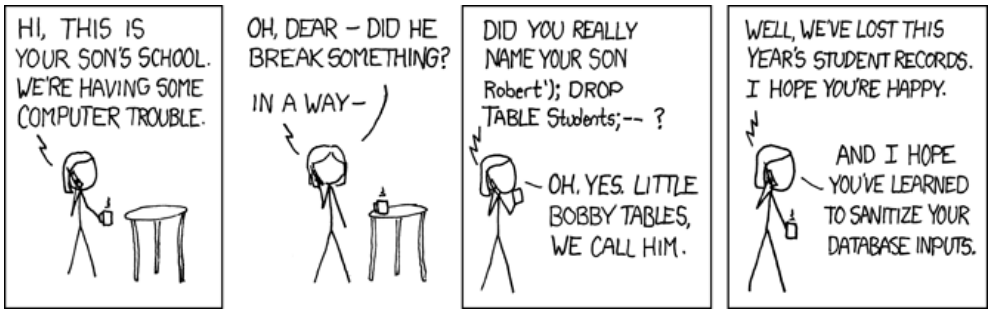


Figure 6.1: https://xkcd.com/327 (2007)

A simple SQL statement to display the students and their grades might be

```
1 SELECT Name, Grade
2 FROM Students;
```

Listing 6.1: `names-grades.sql`

If you wanted to display only the grade of a particular student called Johnny, you would add a `WHERE` clause:

```
1 SELECT Name, Grade
2 FROM Students
3 WHERE Name = 'Johnny';
```

Listing 6.2: `johnny-grade.sql`

If you wanted to offer a web front-end to the teacher so that she could type the name of the pupil in a form and have the query executed on her behalf, rather than asking her to type raw SQL commands, the web front-end would have to collect the new student's name (say Jennifer) from the form and paste it into the previous statement in the place of Johnny, while preserving the single quotes around it, yielding

```
SELECT Name, Grade
FROM Students
WHERE Name = 'Jennifer';
```

Listing 6.3: `jennifer-grade.sql`

Now, what would happen if the teacher typed the name of Little Bobby Tables (that is, `Robert');␣DROP␣TABLE␣Students;--␣`) into the form, and the web front-end substituted it verbatim in place of Johnny between the quotes? The result would be

```
SELECT Name, Grade
FROM Students
WHERE Name = 'Robert'); DROP TABLE Students;-- ';
```

Listing 6.4: `little-bobby-tables.sql`

We can see that this will just cause a syntax error because of the mismatched closed bracket after Robert. Clearly the intention of the devious mum was to close the original statement and start a new one with her destructive payload (deleting the `Students` table), but this only works when the injected code matches the original code. For example, her payload would only be destructive if the school database did indeed have a table called `Students`: if the table were called `Pupils` it would not get deleted. So the mum has to rely on one or more of:

- infallible guesswork (as used for comedic purposes in the cartoon), or
- trial and error, or
- a priori knowledge of the software she is attacking (for example because it is available as open source).

Here the mum's guess is incorrect with respect to the `SELECT` statement we came up with. From the fact that the school phones to complain we surmise that, in the story, the attack actually worked and that, therefore, the school's SQL must have included a bracket somewhere, for example in a statement of the form[2]

[2]Note that actually this SQL statement would have been syntactically correct even without the brackets. And we could have added the brackets around `Name = 'Johnny'` without having to add `AND Year = 3`. We are just making things up as we go along.

```
SELECT Name, Grade
FROM Students
WHERE (Name = 'Johnny' AND Year = 3);
```

Listing 6.5: `johnny-year-3.sql`

When substituting the child's name for Johnny, this would transform into

```
SELECT Name, Grade
FROM Students
WHERE (Name = 'Robert'); DROP TABLE Students;-- ' AND Year = 3);
```

Listing 6.6: `little-bobby-year-3.sql`

which is now syntactically correct and would execute. Note the significant points of the mum's attack:

1. The field must be terminated. Here, this is done by the single quote after `Robert`.
2. The vulnerable statement must be terminated. Here, this is done by the semicolon (always necessary) and by the preceding close-bracket (necessary under the assumption that there was an opening bracket in the original statement).
3. The payload (here `DROP TABLE Students`) must then be inserted as a separate statement.
4. The payload itself must be terminated (here, with the semicolon after `DROP TABLE Students`) in order to be syntactically correct.
5. After the payload, whatever is left of the original SQL statement template must be commented out, otherwise it would cause a syntax error. Here, this is done by the double hyphen plus space (`--␣`) that marks the start of a comment extending to the end of the current line.

6.2 Countermeasures

The mum's advice in the last panel is that the school ought to have *sanitised its database inputs*. What does that mean? If the input routine of the web form had disallowed the single quote character, the attacker would not have been able to perform step 1 of the attack; if the routine had disallowed the semicolon, the attacker would not have been able to perform step 4; and if the routine had disallowed the double dash,

this would have thwarted step 5. However this reactive approach of attempting to forbid metacharacters—or even, for added paranoia, just whitelisting the 26 letters of the Roman alphabet and excluding everything else[3]—is rarely reliable, especially if homebrew: ingenious attackers may find other ways of terminating the substring, particularly when the result is subjected to more than one round of further parsing.

Excluding metacharacters is a rather blunt approach to quoting, which betrays the incompetence of the programmer. The sophisticated and "correct" form of quoting is one in which *any* user input, whatever the metacharacters in it, is transformed and escaped in a way that guarantees it will only be interpreted as data. But is usually difficult to do this with 100% correctness, unless the quoting routine is written by the language developers and maintained by them every time the language is tweaked and the parser is updated.

The preferred countermeasure against SQL injection is to use a *prepared statement*, in which the parametric SQL query is submitted to the database engine, parsed and precompiled, whereas at runtime the parameters are plugged into the "blanks" and the query is executed with the fields as data and, crucially, *without re-parsing* the SQL statement. But the additional programming effort involved, although modest in theory, is still sufficient to deter some programmers from adopting this practice, leading to the persistence of SQL injection vulnerabilities in currently deployed software systems.

Note how the mum's chosen payload in the cartoon is highly destructive: essentially denial of service, or vandalism (availability threat[4]). But it is pretty easy to detect as soon as it happens (everything stops working) and it is relatively easy to recover from, provided that the victim ran a sensible backup regime (perhaps an optimistic thought). A more subtle payload, which just exfiltrated (confidentiality threat) or altered (integrity threat) a specific piece of information, might be harder to detect and difficult or impossible to recover from.

A common idiom in SQL injection attacks is to disable the filter in the `WHERE` clause by making the condition always true, for example by appending `OR 1=1` (or even `OR True`, if the SQL dialect being attacked accepts it). Then, instead of getting the rows matching the originally intended boolean condition, you get all the rows. This is also useful for bypassing parts of the pre-written condition that you don't know how to fulfil, such as "if the password is the correct one": you transform that to

[3] Unacceptably draconian and narrow-minded with respect to such names as `O'Connor` or `Pelé`—not to mention those that do not even use the Roman alphabet in the first place.

[4] Cfr. the brief discussion of Confidentiality, Integrity and Availability in Section 7 on page 89.

"if the password is blabla or 1=1"; the first boolean predicate will be false but the second one will be true and you'll be let in with any password. Try applying these tricks to your SEED lab exercises.

Chapter 7

Passwords

Traditionally, Confidentiality, Integrity and Availability[1] are considered three of the most important properties in security. Given a set of objects in our system[2] and a set of subjects (known or unknown) who might wish to access them, **confidentiality** (the stuff of secret codes, ciphers and access control) is about ensuring that objects may only be read by authorised subjects; **integrity** is about ensuring that only authorised subjects may modify the objects; and **availability** is about ensuring that, whenever authorised subjects wish to access objects, they are able to do so within the agreed/expected service time. All three of these definitions rely on being able to distinguish the authorised subjects from the unauthorised ones who might attempt to impersonate them—which we might take as a definition for **authentication**. We thus see that authentication is an implicit prerequisite for all three of the foundational CIA security properties.

The subjects to be authenticated might be humans, roles, organisations, even machines; and so might be the entity performing the verification. The authentication might be unidirectional, from a prover to a verifier, or bidirectional, where each of the two entities proves its identity to the other. There might be a malicious adversary in the middle, observing all communications and later replaying them selectively, perhaps with subtle tweaks, in order to impersonate a prover to a verifier. Protocols have been devised to perform authentication in almost all possible combinations of scenarios. Flaws have been discovered in many of

[1] Sometimes referred to as "the CIA triad", but not related to the US spy agency with the same acronym.

[2] We mean "system" in the most general sense: do not merely think of a computer but a network, an organisation, even a country or a coalition of countries. And don't include just machines in it but also the people who administer them and those who use them. And their customers. And their files. And the individual pieces of information in them. And...

them, and sometimes patched later. Formal verification tools have been created to prove or disprove the correctness of proposed authentication protocols. Protocols that were proved correct have been successfully attacked nonetheless, by violating the modelling assumptions of the formal tools used to perform the verification. Authentication is a vast subfield of security and, no matter how much we say, we will not be able to do justice to many of its facets—and this would still be true even if we devoted the entire course just to this topic.

Traditional wisdom on authenticating humans is that the verifier should check "something you know, something you have or something you are". The feature to be checked should be unique to the verifier and hard to replicate by would-be impersonators.

In this chapter we'll shine a spotlight on passwords, the prototypical "something you know". Passwords are still widely used today as a technology for user authentication to a remote machine, despite having been considered obsolete and flawed for that purpose for more than two decades. We'll examine how passwords work, how they fail, what has been attempted to replace them and why the alternatives have not been able to eliminate passwords yet.

7.1 How passwords work

The typical user interaction pattern for logging into a remote website consists of two questions: *username?* (who do you claim to be?) and *password?* (now prove it). Whether defensible or not, the underlying assumptions include:

1. nobody other than you knows your password

2. nobody is able to *guess* your password

3. the website is able to verify whether the password you supply is the correct one.

Both the first and second assumptions are implausible when the password is chosen by the user. Conversely, when the password is a random string assigned by a computer (which, for sufficiently long strings, substantiates the second assumption), it is unlikely to be remembered and it usually gets written down. This might in turn lead to violations of the first assumption.

7.1.1 Hashing

In the old days, to make the third assumption hold, the website remembered the plaintext password as well. Besides being a straightforward violation of the first assumption, this practice allows an attacker to learn the password to all user accounts as soon as she breaks into the website. In the Eagle pub in Cambridge, in the early 1970s, Roger Needham (1935–2003), later to be the PhD advisor of my PhD advisor Ross Anderson (1956–2024), invented a better way of doing it: the password is fed through a cryptographic hash function, yielding a scrambled up *digest*. The digest is easy to compute, given the input, but it is computationally infeasible to recover the input from it[3]. Then, only the digest (not the plaintext password) is stored on the server, next to the username. On authentication, the prover supplies the password, the verifier hashes it (without storing it) and checks the result against the stored digest. The attacker who cracks into the server may steal the digests but may not easily recover the original passwords from them. This clever invention spread widely and eventually became almost universally adopted by subsequent multi-user operating systems, starting with Unix (though not by the many websites that reinvented the login wheel with no awareness of prior art—you'd be surprised how many clueless websites still store passwords in plaintext to this day).

7.1.2 Salting

Although the attacker who captures the list of digests will not be able to derive the password from the digest because the hash function is not invertible, she may attempt to brute-force the password and use the leaked digest to verify her attempts. The most naïve attacker tries all possible passwords, leading to those unrealistically optimistic claims that an 8-character password including upper and lower case characters and digits will require $(26 + 26 + 10)^8 \approx 200$ trillion attempts. But such a clueless attacker only exists in the imagination of the people who circulate these misguided statistics. Any attacker with enough brain cells will start by trying the passwords that people actually use—things like "123456", "qwerty" and "password" (search the web for "most common passwords" to

[3]The cryptographic hash function is designed to be easy to compute but infeasibly hard to invert. To the extent that it can consistently map strings of arbitrary length into fixed-size digests, the hash function is also *theoretically impossible* to invert, because it is not a bijection: for the pigeonhole principle, there will exist at least some digest (in practice almost all of them) with infinitely many preimages. But when we say "hard to invert" we mean that, given a digest, it is computationally too expensive to derive *even just one* of its possibly infinite preimages.

download longer lists), then person names, pet names, words from the dictionary, leetspeak spelling variations on the above and so forth. The total number of these plausible candidate passwords is tiny: perhaps 8–10 orders of magnitude smaller than those alleged 200 trillions, and yet it usually still cracks *some* non-negligible proportion of the available accounts. This technique (trying *likely* passwords) is known as a **dictionary attack** and is much more efficient than a **brute force** attack which just tries all possible passwords.

An important mindset shift is that, spearphishing operations aside[4], the attacker is not interested in cracking a particular account but merely in cracking *some* account—that is to say, the ones with the easiest-to-guess passwords; these can be raided for whatever assets they contain and then maybe also exploited for privilege escalation, as a first foothold on the multi-user machine under attack. Therefore, when the attacker hashes "123456" and looks for its presence in the captured digests, if she finds a match she is happy to have cracked an account but does not care whose it was.

Note also that, if two distinct users choose the same weak password, then, once those identical passwords are hashed, these users will have the same digest in the credentials file. Thus, non-unique digests are a strong hint to the attacker that their common preimage must be an easily guessed password (not a hard-to-guess high-entropy one), because two people independently came up with it. The insider threat is perhaps even worse: if Alice, who legitimately has an account on the machine, notices that Bob has the same digest as her, she knows they both have the same (weak) password—and she can now log into Bob's account! This is one of the reasons why, in Unix, the digests were moved from `/etc/passwd`, where they were originally stored in the second field, to a new file called `/etc/shadow`. The `/etc/passwd` file had to remain world-readable because the uid-to-name mapping contained therein needed to be consulted by a variety of unprivileged programs[5], but the new `/etc/shadow` could be made unreadable by anyone other than `root`.

The smart attacker of a system where passwords are hashed will observe that she may start cracking passwords even *before* penetrating the server: she may as well hash candidate passwords in her spare time anyway, and save the (password, digest) pairs in a large lookup table[6], investing months and years in this effort if need be. And then, as soon as she captures (or buys, or anyhow acquires) a file of digests, she can crack it on sight by looking up the corresponding passwords in her table.

[4] Where a specific high-value individual is targeted.

[5] E.g. `ls -l` to show file owners.

[6] Keyed by digest, of course, not by password.

In other words, both the dictionary attack and the brute-force attack (up to a predefined password length) can be sped up to constant time (!) by investing a sufficient amount of precomputation and storage into the construction of a reverse lookup table that maps digests back to the passwords they come from. This is pretty bad news for honest users of the system.

To defeat this attack optimisation, in the late 1970s Unix security architects Bob Morris and Ken Thompson introduced the practice of **salting** the passwords before hashing them[7]. For each password, a random string is generated (the salt), and it is the concatenation of the password and salt that gets hashed to form the digest. The password file then stores, next to the username, the digest and also the salt that was used to obtain it. To verify a password supplied by a prover, the verifier looks up the salt in the password file, concatenates it with the candidate password, hashes the lot and checks whether it matches the digest. The attacker who exfiltrates the password file is still able to do the same, or course, so the presence of the salt does not strengthen the system against attackers who try weak passwords (dictionary attack); but at least the salt stops attackers from cracking those weak passwords *on sight*, with just one lookup of a pre-computed table. Salting exponentially increases the cost of computing and storing that lookup table. A salt of k bits increases the computation and storage requirements for the attacker by a factor of 2^k, which is a wonderful trade-off for the defenders (inexpensive defense yielding greatly increased attack cost).

As further protection against mass cracking of passwords, the technique of **key stretching** consists of taking the password and salt and then hashing them repeatedly (say 1000 times[8]), so that the digest-generating function becomes $h^{1000}()$ rather than just $h()$, and hence is 1000× slower[9]. Although the computation is still almost instantaneous when verifying a user's password in the course of legitimate operations, it will now take a thousand times longer for attackers to attempt to crack the easy passwords in an exfiltrated `/etc/shadow` file (or to precompute the lookup table) on their dedicated bank of GPUs (as these people do).

[7] Robert Morris and Ken Thompson. 1979. "Password security: a case history". *Commun. ACM* 22, 11 (Nov. 1979), 594–597. https://doi.org/10.1145/359168.359172

[8] Actually the number of iterations may even vary from one password to another, provided it is stored with the digest alongside the salt.

[9] And can't be sped up by parallelisation because each intermediate result depends on the previous one.

7.1.3 Precomputed hash chains

Although salting *with a sufficiently large salt* is an effective countermeasure against precomputation of hashes, some very widely deployed systems such as Microsoft Windows nonetheless shamelessly continue to use unsalted passwords even as I update these notes in 2026[10].

It thus became advantageous for attackers, given the economies of scale involved, to build ever-larger lookup tables, covering more and more of the password space: the computation of those tables is amortised over the hundreds of millions of systems that can be attacked, and conversely the attackers can perfectly parallelise the computation of the lookup tables since there are no dependencies between the entries. With this strategy, and by distributing the workload in parallel among many cooperating volunteers, it becomes feasible for attackers to pre-compute the digest not just of commonly used passwords but even of *all* possible passwords up to a certain length.

Clearly, for any exhaustive brute-force lookup table, the space needed grows exponentially with the length of the passwords[11] and thus inevitably exceeds, at some point, the capacity of commonly available storage media (hard drives etc). **Precomputed hash chains** are an ingenious time-space tradeoff construction that reduces the space cost to store a full lookup table, at the time cost of having to repeat some part of the computation on every lookup. How do they work?

Consider the set P of all possible passwords constructed according to certain well-specified rules[12], and the set D of all possible digests that the chosen hash function $h()$ might output[13]. The signature of the hash function is $h : P \to D$ and the lookup table contains one $(p, d = h(p))$ pair for each $p \in P$, with d being the key and p the value ("give me a digest d and, with one lookup, I'll tell you a password p that hashes to it"). Clearly, for 128-bit digests, the table would have $|D| = 2^{128}$ entries and we would never be able to afford enough time to compute it, let alone space to store it—and even worse for hash outputs longer than 128 bits. However we are not really interested in a preimage for every possible

[10] The notorious Microsoft Lan Manager (1987) and NT Lan Manager (1993) hashing methods were unsalted, and continued to be supported in Windows for many years for backwards compatibility even after the corresponding products had become obsolete and this vulnerability was being exploited in the wild. Microsoft is currently in the middle of a multi-year plan to deprecate NTLM but it still continues to use unsalted NT hashes for the password storage of local accounts.

[11] The number of possible distinct passwords is $\sum_{p=0}^{m} c^p$ where c is the size of the character set (e.g. $26 + 26 + 10 = 62$ for upper, lower and digits), p is the number of characters in the password and m is the maximum allowed length for a password.

[12] For example: up to 8 characters of lowercase, uppercase and digits.

[13] For example: all the possible strings of 128 bits.

digest in D; we are only interested in the preimages of the digests that were obtained by hashing passwords in P, of which there are only $|P|$. Typically, $|P| \ll |D|$, and by very many orders of magnitude.

Assume $|P|$ is still sufficiently large that the corresponding table would exceed your available storage capacity; but assume also that, since the task is fully parallelisable, you and your internet friends together have the computational capacity, given enough time, to compute each entry in the table. The following construction (precomputed hash chains) allows you to replace the large table with a much smaller one, at the price of increasing the time required for each lookup[14].

To arrange the (p, d) pairs into long hash chains, we introduce a reduction function $r : D \to P$ that maps digests back into the space of passwords[15]. Then a chain is obtained by starting from a password and applying alternatively $h()$ (to get the password's digest) and $r()$ (to map the digest back to a password). This alternation is repeated a predetermined number of times, say l; then $h()$ is applied one last time and then only the initial password p_i and final digest d_f are retained to represent the chain in the compressed table, with d_f as the key and p_i as the value. Note that the chain-specific value p_i (initial password) and the global parameter l are sufficient to reconstruct the whole chain[16].

Ideally, enough such chains are generated (and their endpoints retained) to cover all the passwords in P. Practically, though, since nothing guarantees that every new chain will contribute $l + 1$ new passwords (*new* in the sense of never having been seen before in other chains), you may instead settle for covering a large fraction of P, rather than all of it.

How to retrieve the password preimage of a given digest d from the compressed table? We first establish which chain d appeared in. Does d match any of the final values d_f?[17] If yes, d obviously appears in that chain as its last value. If not, compute $d' = h(r(d))$ and see if *that* matches any of the d_fs. If not, keep reducing and hashing the partial result for up to l iterations. By then, at some point the hash output *must* have matched one of the d_f if the original d appeared, as assumed, on at least one of the chains.

[14] The factor by which the lookup time increases will be roughly proportional to the factor by which the size of the lookup table decreases.

[15] Note that $r()$ is not expected to be the inverse of $h()$ since, by definition of cryptographic hash, it is computationally infeasible to invert $h()$. Instead, $r()$ is just a function from digests in D to passwords in P, but it doesn't promise to map a digest to a password that would hash to it.

[16] The parameter l is basically the length, although pedants will note that, to be precise, the chain contains $l + 1$ passwords and $l + 1$ digests. Not that the difference matters much.

[17] We can answer this easily in constant time, since the compressed lookup table is keyed on those final digests, just like the raw table was.

Having identified a chain that contains d, we still don't know the password that hashes to d. To find it, we simply rebuild that chain from its starting point p_i (given by the lookup table) until we get to d, and then we look at the origin of the preceding $h()$ link, namely the $p \in P$ such that $h(p) = d$. That p is the sought password.

This "precomputed hash chains" construction (originally due[18] to Martin Hellman (1945–) of Diffie-Hellman fame, one of the inventors of public key cryptography and Turing Award winner) offers the core idea of the space-time trade-off. The storage requirements for precomputed hashes are reduced by a factor of l, but looking up the password corresponding to a given digest now requires the computation of about l hashes and l reductions rather than just one single lookup in the uncompressed table.

One problem with this construction, however, is collisions: multiple passwords might in theory hash to the same digest but also, and rather more likely, multiple digests might reduce to the same password. The larger the table, the greater the probability of this happening. When two different chains collide, then, from the point of collision onwards, the chains, albeit distinct, produce the same sequence of password and digest values. When a single chain collides with itself, then a cycle results. In both cases, the naïvely optimistic assumption that "k chains of length l cover kl passwords" is falsified: they may cover many fewer. In other words, the space reduction factor from the raw table to the compressed one may be significantly lower than l, even though we still pay the same (or in fact worse[19]) slow-down in lookup time.

To limit the probability of collisions, Hellman suggested limiting the size of the lookup table and, if necessary, starting new tables, each with a distinct reduction function. But this comes with additional time costs, because several tables must then be searched during lookup.

[18] Martin Hellman, "A cryptanalytic time-memory trade-off". In *IEEE Transactions on Information Theory*, vol. 26, no. 4, pp. 401–406, July 1980. http://www-ee.stanford.edu/~hellman/publications/36.pdf.

[19] Imagine digest d_1 appears in the chain starting at password p_1 and digest $d_2 \neq d_1$ appears in the chain starting at p_2. Imagine further that those digests collide and reduce to the same password: $r(d_1) = r(d_2) = p_3$. Then, when seeking the preimage of d_1, we'll reduce it (to p_3) and hash it and reduce it till we get to one of the digests among the keys of the compressed lookup table. But, since the two chains collide, the sequence of hashes and reductions from p_3 onwards will be the same in the two chains. Assume that d_2 is closer to the end of its chain than d_1 is to the end of its own; then the sequence of hashes and reductions will lead us to the endpoint of the chain starting at p_2. If we reconstruct that p_2 chain from the start, seeking d_1, we eventually discover that d_1 was not in it: a so-called *false alarm*. Then we must continue reducing and hashing past the end of that chain, continuing the subsequence. Eventually (unless there was another collision), we'll reach the endpoint of the p_1 chain. And then we'll have to restart *that* chain from the beginning to find the preimage of d_1.

7.1.4 Rainbow tables

An elaborate variant on precomputed hash chains, due to Philippe Oechslin[20] and called **Rainbow tables**, which many people mention but actually confuse with Precomputed hash chains, handles the problem of collisions by using a different reduction function at every step of the chain. Then, even if two chains collide on one value, the subsequent values will not overlap in the two chains, unless the collision happens at exactly the same step in both chains. The lookup procedure becomes a little more complicated (see the original paper) but overall there is a gain. With fewer collisions, for the same chain length l, rainbow tables need fewer chains to cover (almost) all passwords in P; or, alternatively, for the same number of entries (same storage space) and collision rate, they cover P with shorter chains, yielding a reduced worst-case lookup time.

Rainbow tables, being more efficient, are what attackers have actually been using against real-world vulnerable systems such as the cited M$ LAN Manager. Since 2006 the website `https://freerainbowtables.com` has coordinated the distributed computation and subsequent dissemination of rainbow tables for a variety of character sets, password sizes and hash functions including MD5, SHA-1, Microsoft LAN Manager and NT LAN Manager.

A salt of sufficiently large size makes even rainbow tables infeasible. The salt introduced by Unix in the 1970s was 12 bits long—increasing the attacker's precomputation and storage costs "only" by a factor of $2^{12} = 4096$, which is within the realm of what the rainbow tables construction could compensate for (by making chains 4096 entries long). Modern alternatives such as `bcrypt` use 128 bits of salt, for a storage increase factor of $2^{128} \approx 3.4 \cdot 10^{38}$, making rainbow tables utterly useless. The login passwords of modern Unix-like operating systems are protected in this way and are thus not vulnerable to lookup tables or rainbow tables, but unfortunately the same cannot always be said of application-level passwords or website passwords, which are sometimes stored with an unsalted MD5 hash, or with reversible encryption—or sometimes even in plaintext. The website `http://plaintextoffenders.com` (now apparently down, but look it up on the Wayback Machine) used to maintain a hall of shame—an eye-opening reminder about programmer incompetence.

[20] Philippe Oechslin, "Making a Faster Cryptanalytic Time-Memory Trade-Off". *CRYPTO 2003*, Springer LNCS 2729. pp. 617–630. `https://infoscience.epfl.ch/record/99512/files/Oech03.pdf`.

7.2 How passwords fail

The main problem with passwords is aptly summed up in a Mikko Hypponen tweet from 2011:

> Overheard password advice "Pick something you can't remember, then don't write it down"

Sometime in the 1960s or 70s, when most ordinary people had never seen a computer and the others had at most a single account, it might have been reasonable to ask those rare computer users to remember one very important login password. But nowadays, when people have to juggle hundreds of accounts, it is unreasonable to demand that they remember a different unguessable password for each. Yet, if they don't, they get blamed for making the system insecure. This adds insult to injury and is fundamentally unfair. Security professionals have an ethical duty to do better.

Each of the demands from narrow-minded security administrators has, arguably, some justification. Passwords, they say, should not consist of names, dates or familiar information otherwise they could be guessed. Passwords should include mixed-case, symbols and digits otherwise the whole password space could be brute-forced. Passwords should be memorised rather than written down otherwise they could be discovered by attackers. But the *aggregation* of these demands makes them ridiculously inappropriate even if they might be individually defensible, because the intersection of all these constraints is the empty set! The techies who come up with such security policies seem incapable of seeing the big picture. They should not be allowed to get away with just dumping the problem on the users: they must offer a solution that achieves an acceptable level of security but without imposing an unreasonable burden on the users.

Clearly, users will not be able to comply with a set of mutually incompatible rules: some rules will have to be violated. One of the most common coping mechanisms is for users to reuse one password on several sites. The problem with this practice is that, when one site is broken into and its password file is exfiltrated and cracked, the other sites where the user reused the same password are now also at risk. Some sites will have lower security than others[21] and thus the security of the set of accounts protected by the same password is bounded upwards by that of the least secure of them all.

[21] Indeed some sites might even be run by crooks in the first place, who store passwords in plaintext *and make a point of reading them* and trying them on other websites where the victims might also have accounts.

This leads to the recommendation to use a different password for every site (in itself a good idea). But this, in turn, is incompatible with the request that all passwords be remembered by heart rather than written down or stored. Something has to give. The least defensible of these directives is this last one (memorise all passwords), which clearly cannot scale to hundreds of accounts. Another indefensible directive is that of choosing something memorable and yet not guessable by attackers. In the age of social media and of GPUs with billions of transistors, this is a contradiction in terms: most passwords that a non-security-nerd can remember can also be guessed, particularly if the attacker seeds the search with personal data about the victim.

Florêncio, Herley and Coskun[22] have insightfully argued that system administrators who impose draconian restrictions on the passwords that their users may choose would instead do better to secure the server that holds their password file. The reason for that is because the threat of brute-forcing the easy passwords is only realistic in an *offline* attack where the attackers have the password file with the digests and they can try candidate passwords as many millions of times per second as their bank of GPUs allows them. Whereas, in an *online* attack, a mass cracking operation would quickly generate an abnormal number of failed logins that ought to alert the site administrators to the attack. This greatly limits the rate at which attackers may try password guesses in an online attack[23]. Thus, if the sysadmins harden their system and disallow the exfiltration of the password file, they force the attackers onto online attacks only, at which point it is no longer necessary to require users to adopt the kind of super-high-entropy passwords that would be needed to resist an offline attack.

7.3 Why are passwords still with us

> "There is no doubt that over time, people are going to rely less and less on passwords. People use the same password on different systems, they write them down and they just don't meet the challenge for anything you really want to secure."

The above quote was from Bill Gates's presentation at the 2004 RSA

[22] Dinei Florêncio, Cormac Herley and Baris Coskun, "Do strong web passwords accomplish anything?", *Proc. HotSec 2007*. http://cormac.herley.org/docs/doStrongPasswordsAccomplishAnything.pdf

[23] It still makes sense to disallow the most commonly reused passwords, though, because if a naïve user picked 123456 as his password then it won't take very many tries for the attacker to guess that.

security conference. So why, more than two decades later, do we still have to juggle more passwords than ever?

Innumerable alternatives to passwords have been proposed. I am quite familiar with this problem, having worked on it for years. I proposed my own solution, a handheld password manager called Pico[24], and even started a company around it. I led the team that wrote one of the most highly cited papers in this field[25], which established a framework for comparing password replacement schemes on the three axes of security, usability and "deployability", our neologism to describe the ease and convenience with which a technology could be deployed. We thoroughly analysed 35 alternative schemes along 25 criteria and noted that, although many alternative schemes were stronger than passwords on the security front[26], and some were better on the usability front, no scheme was superior to passwords on the deployability front. The friction necessary to overcome the transition costs was the reason why we expected passwords not to go away anytime soon.

The experience with my Pico startup, Cambridge Authentication, which failed early, taught me valuable practical lessons on security usability. Even though every user we met professed to hate passwords and to love us for developing a more usable replacement, they had all evolved their own coping strategies to which they were dearly attached. They were not willing to endure the friction of moving to a new system, even if theoretically easier to use and more secure, unless it solved the password problem in all possible circumstances. A prototype that only worked for an initial subset of cases was not appealing, because they'd have to use both the new and the old system in parallel. This gave an almost unbeatable advantage to the incumbent technology of passwords, particularly as the more radical alternatives required changes both at the user end and at the server end[27]. Site operators would not consider supporting a new

[24]Frank Stajano. "Pico: No more passwords!". In *Proc. Security Protocols Workshop 2011*, Springer LNCS 7114. https://www.cl.cam.ac.uk/~fms27/papers/2011-Stajano-pico.pdf.

[25]Joseph Bonneau, Cormac Herley, Paul C. van Oorschot and Frank Stajano. "The Quest to Replace Passwords: A Framework for Comparative Evaluation of Web Authentication Schemes". In *Proc. IEEE Symposium on Security and Privacy 2012*,http://www.cl.cam.ac.uk/~fms27/papers/2012-BonneauHerOorSta-password--oakland.pdf

[26]As could have been expected from the fact that proponents of alternatives to passwords are usually members of the security community.

[27]Our initial design was based on public key cryptography, with a different key pair for each site. It totally protected sites from those disastrous breaches in which millions of user accounts get compromised (because exfiltrating the public keys does not allow the attackers to impersonate the users, unlike what happens with the crackable password hashes) but it would have required incompatible changes in the back-end.

technology unless most users had adopted it, and conversely users would not bother adopting a new technology unless they could use it with most sites and platforms. This validated our paper's prediction that deployability would be crucial for the success of any new password replacement scheme.

One reason why passwords have always been the default go-to method for user authentication despite their undeniable usability shortcomings is what economists call a *moral hazard*[28]. It's the website (not the users) that decides how to authenticate users, but the website does not bear any of the costs (the usability pain) of choosing the passwords over other methods. On the other hand, for the website, passwords are the easiest technology to deploy: everybody understands how to use it, it's supported by all client platforms and, crucially, the marginal cost per additional user is essentially zero, which is not the case for any schemes involving physical tokens.

As my coauthor Cormac Herley once insightfully remarked: how could Facebook have reached one million users before its first injection of capital, if it had adopted an authentication technology with a non-zero cost per user?

7.4 Alternatives to passwords

Going back to "something you know, something you have, something you are", one might wonder how hardware security tokens (something you have) and biometrics (something you are) would fare against passwords.

Authentication tokens have the potential of offering better security: they might have secure storage for user credentials and key material, as well as processing power for running cryptographic operations and authentication protocols. They generally fare poorly on usability, especially when they don't talk directly to the back-end over some communication network but instead require manual user interaction to transcribe authentication codes between a tiny screen on the handheld token and the user's computer. But actually the main usability complaint about hardware tokens is simply the need to carry them around: to a first approximation, only the smartphone has provided sufficiently high utility to its user to have earned the privilege of being carried everywhere, on a par with house keys and wristwatch. Dedicated security tokens do not reach

We migrated to slightly less secure schemes requiring fewer or no modifications to the back-end, but we were not able to cover all possible use cases in our initial prototype, which made switching unattractive for the users.

[28] "A situation where an economic actor has an incentive to increase its exposure to risk because it does not bear the full costs of that risk" (Wikipedia).

that bar, particularly when it's one token per account rather than a single universal token for all accounts. In many cases, tokens are deployed when the system administrator requires greater security and imposes the tokens on the user base without leaving them the choice and ignoring usability complaints ("I forgot the f#@!%&$ token at home and so today I could not log in"). Smartwatches, bought and worn voluntarily by users as fashion/tech items, have a better chance of serving as security tokens, particularly when the same company is in the privileged position of designing and manufacturing all three of the smartwatch, the computer it talks to and the operating system that accepts the smartwatch as an authentication token instead of a password. Smartphones, for that matter, have largely become security tokens, precisely because users carry them around voluntarily and look after them with care. Of course physical tokens may be lost (which denies service to their owner) or stolen (which, besides the denial of service, may result in impersonation by the thief, unless the token itself must be unlocked before use, which in turn might reduce its usability).

Biometrics, if properly implemented, have the potential of offering high usability, insofar as the user does not have to either remember or carry anything. Regarding security, several considerations are in order. Firstly, replay attacks. Biometrics are not secrets: they are observable features of the individual. Authentication through biometrics is about recognising those features as yours, but this must be coupled with a liveness check to assess that the features were sampled from a person, as opposed to having been replayed to the sensor (gummi finger for fingerprints, photograph for iris scan or face recognition, recording for voiceprint recognition etc). Where the liveness check is carried out with a challenge-response protocol ("Repeat the following random word, so I know it wasn't a recording"), we are no longer in the realm of do-nothing technology and usability suffers. But, more importantly, the requirement for liveness check makes biometrics utterly unsuitable for remote unsupervised authentication, which is the standard "log into a website across the network" use case. The bank at the other end of the cloud has no guarantees that the finger that went on the fingerprint reader, even though it matched the fingerprints on file, is that of the accont holder as opposed to being a gummi replica (or, in a more gruesome version, the chopped-off finger). Secondly, biometrics cannot be revoked or changed. If a back-end is compromised, your fingerprints/irises/whatever will leak. And the fact that you have only one set (of irises, of fingerprints etc) is also a privacy concern, because your interactions with various verifiers (banks, shops, drinking holes, gambling joints, sex clinics and anything you might wish to be discreet about) can be linked to each other, in a way

that would not be possible if you used different credentials with each. A good niche for biometric authentication is unlocking a local device that you own, such as your smartphone, where relinking is no longer a concern. You will still have to worry about replay/impersonation attacks if the device is stolen, but the convenience of not having to type a PIN several times a day may outweigh that concern if theft is an infrequent and unlikely occurrence.

A systemic approach to address the proliferation of passwords is the so-called Single Sign-On—an arrangement whereby you authenticate once to some kind of authentication service provider (ASP) and thereafter the individual sites talk to that ASP directly and let you in without any further interactive challenges. An insightful paper by Pashalidis and Mitchell[29] classifies SSO systems along two axes: whether the ASP resides on the user's machine (the case they designate as "local") or on some remote server in the cloud (which they call "proxy-based") and whether the ASP runs a proper cryptographic authentication protocol with the verifiers (which they call "true SSO") or simulates the sending of passwords for backwards compatibility (which they call "pseudo SSO"). This yields four quadrants. The various options have complementary tradeoffs and this taxonomy is a good way to think about them. In practice, the problem with SSO systems is that "single" is usually wishful thinking: one SSO will log you into a variety of systems, yes, but you will still have to log into many others not covered by that SSO where you'll have to authenticate in other ways, probably by retyping a password. We observe that the password manager in the browser, one of the most sensible approaches to cope with the unending proliferation of passwords and one of the most highly rated schemes in our comparative analysis of password replacements, fits into this taxonomy as a "local, pseudo-SSO system".

We shall have more to say about security usability in Chapter 10 on page 129, but observe the different attitudes of the policy-setters when they have a captive audience versus when their users may choose to go elsewhere: Florêncio and Herley[30] perceptively observed that it is primarily in the second case that we witness greater attention for usability than for security, in recognition of the fact that annoyed users may vote with their wallet and spend their money elsewhere. Thus the password

[29] Andreas Pashalidis and Chris J. Mitchell. "A Taxonomy of Single Sign-On Systems", In *Proc. Australasian Conference on Information Security and Privacy* (ACISP 2003), pp 249–264, Springer LNCS 2727, 2003. http://chrismitchell.net/atosso.pdf.

[30] Dinei Florêncio and Cormac Herley. "Where Do Security Policies Come From?". In *Proc. Symposium on Usable Privacy and Security* (SOUPS 2010). https://cups.cs.cmu.edu/soups/2010/proceedings/a10_florencio.pdf.

security policies that corporations impose on their employees (uppercase, lowercase, at least one symbol and a number...) tend to be more draconian and inconvenient than those they offer to their potential customers. Extending from that observation, we note that websites that want their visitors to stay logged in realise that visitors would rather avoid the site than having to type a password at every visit. Such websites therefore drop a long-lived cookie in the visitor's browser at first login that keeps the visitor logged in for months without annoying them further. This assumes that the endpoint (the visitor's laptop or phone) is trustworthy—a reasonable trade-off compared to forcing a new login at every visit, which would most likely deter the visitor from returning.

Indeed the greatest improvements to security usability (in particular with respect to ameliorating the situation in comparison to passwords) are likely to come from companies with a profit-based reason for alleviating the pain that their customers experience. E-commerce companies prefer to dial down the security and absorb the losses if this makes customers more comfortable with visiting the site and makes them spend more time shopping. Even banks are now moving on from their allegedly super-secure dedicated login tokens to smartphone apps that do the same thing, but on a device that the user carries voluntarily. When companies choose to allow their users to log into their websites with weak passwords, and they also give them a long-lived login cookie so that they only have to type the weak password once a month, they compensate by doing more security work in the back-end. The password is now only treated as one of several hints[31], others being the above-mentioned long-lived login cookie, the IP address from which the user is logging in, the browser, the machine, the geolocation, the time of day and so forth. Each indicator is assessed against its expected and historical values and flags are raised when too many indicators signal an anomaly. At that point the user is challenged to provide further evidence of the genuineness of the login: for example, she may be asked to respond to a challenge sent through another channel such as email or phone. These extra checks (typically triggered by a trip abroad or a change of laptop) may be disruptive, but they are only invoked when anomalies are detected. When a login does not look suspicious because it follows the usual pattern, the user is allowed in with minimum fuss.

A promising emerging technology for replacing passwords is the so-called Passkey (WebAuthn), for which my own Pico might count as prior

[31] Joseph Bonneau, Cormac Herley, Paul C. van Oorschot, Frank Stajano. "Passwords and the Evolution of Imperfect Authentication". *Comms ACM* 58(7):78-87, July 2015. http://www.cl.cam.ac.uk/~fms27/papers/2015-BonneauHerOorETAL-passwords.pdf

art. We might class it as a "local, true SSO". The local device (your laptop or phone) is trusted and becomes the SSO. It stores a distinct key pair for every verifier. The credentials file of the remote website (the verifier) only stores the public key for each user; therefore it cannot be used by the crooks for recovering user credentials even if they manage to exfiltrate it and process it offline. To log in, the local device (the prover) runs a cryptographic protocol to prove ownership of the private key to the verifier, but the private key never leaves the local device. The local device must be unlocked before it may use a stored private key, but this is typically achieved through biometrics and the user no longer has to remember or type a password.

Chapter 8

Cross-Site Request Forgery

Textbook

Study Chapter 12 in W. Du, *Computer Security* 3rd ed.

SEED labs

Complete the following lab: Web Security | Cross-Site Request Forgery Attack Lab (tasks 1, 2, 3, 4; optionally 5).

8.1 Web cookies

CSRF exploits cookies so let's first recap the essentials of how cookies work.

HTTP (Hyper Text Transfer Protocol), the fundamental protocol underlying the web, started out as stateless: the client sends an HTTP `GET` request to the server, with semantics of "send me the web page at this URL", and the server sends back a response with either the requested page or an error message. There are ways to add parameters to a request, but distinct requests are independent of each other.

Seeing the web as a platform on which to build distributed applications, developers soon recognised the need to create *sessions*: in the case of an online store, for example, the customer selects a fluffy toy on one web page, a lollipop on another, and then, when she visits the checkout page, she must see a list with both of these items in her shopping basket.

The technical mechanism (introduced by Netscape in 1994) to add sessions on top of HTTP is the *cookie*: when the server responds to

a request, alongside the response it may also send the client a cookie, which is simply a small piece of text with arbitrary content chosen by the server, and an expiration date. This is basically the server telling the client "Please remember this for me". The convention is that the client will attach the cookie(s) it received to any subsequent request it makes to the same server (provided they have not expired yet). Of course the client is free not to return the cookies, or to return only some cookies, but functionality might then be correspondingly degraded, as the server loses track of what the client had done in previous interactions (e.g. selecting the fluffy toy and the lollipop).

One possible approach to use cookies to implement the previous example would then be for the website to send back a "shopping basket" cookie that listed all the items the customer had selected so far; visits to subsequent product pages would accumulate further items into this cookie as appropriate, whereas visiting the checkout page would list all the items in the cookie and offer the option to pay for them. In this approach the state of the transaction is kept client-side, which relieves the server of the burden of keeping track of it, but the state can be manipulated by the client.

The complementary architecture is to make the cookie nothing more than a session identifier, and to store all session state at the server, keyed by that session ID. Then the client that returns the cookie to the server can pick up where it last left off in the previous interaction, but it is not at liberty to change the state between one interaction and the next. If the server does not trust the client (and why should it?), this is clearly a superior architectural choice from a security perspective[1]. (Way back

[1]Alternatively, the server could still store the state client-side but encrypt it and MAC it, to prevent the client from reading or silently modifying it. This solution has the advantage of greater scalability for the server because every additional client comes with its own storage.

Incidentally, "to MAC an object" is informal shorthand for "to apply a MAC to an object". OK, and what is a MAC? A Message Authentication Code can be described as a keyed hash. It is a function that takes as input an arbitrarily long string, as well as a secret key of fixed length, and returns a digest of fixed length that is dependent on all bits of both the string and the key. As with the hash, it must be computationally infeasible to invert the MAC and to derive any inputs that would yield a given digest. The difference is that, while anyone can verify whether a given string hashes to a given digest, with the MAC this check can only be performed by the holder of the key.

In our example above, if the server tried to protect the integrity of the state against modifications by the client by appending a hash (and making a point of verifying it before trusting any state sent back by the client), then there would be nothing to stop the client from recomputing and appending a new hash so that it would match the maliciously modified state. But, if the server uses a MAC instead, then the client cannot generate it without possession of the key that only the server knows.

then, an attack on some early and naïve websites that kept the shopping cart client-side was to reduce the prices of the items in the cart before paying for them.)

A session could be an authentication session ("Dear webmail site, I have logged in, so in all subsequent requests please show me my own emails until I log out"), in which case the first request to the site (login) would involve some authentication protocol, and the response to that would contain an "authentication cookie", possession of which would prove to the server that the client represents the previously authenticated user.

It is apparent that stealing authentication cookies would allow the thief to impersonate the victim. One countermeasure against cookie theft is the introduction of the "Secure" attribute for cookies: if a cookie is marked as secure, the client will only send it back in an HTTP request if the request is transmitted over a secure channel (such as HTTPS).

8.2 Phishing

Before delving into CSRF, another little aside about a different and much more common kind of web impersonation fraud.

Phishing is a very widespread but also very low-tech fraud in which the attacker reproduces the login page of a website and then sends the victim an email with a link to that website. Through a variety of social engineering tricks (appealing to greed, authority, time pressure, scarcity and other principles that we'll be examining in section 10.3 on page 135) the email persuades the victim to log in to the fake website, where the crook then captures his username and password.

No great technical sophistication is required on the part of the attacker. The HTML of the login page of the target website can essentially be copied verbatim. And there is no need to build any more than a Hollywood façade because the website can simply pretend to crash after the credentials have been acquired.

The front-line technical countermeasure against phishing is `HTTPS`, which consists of running HTTP over TLS (an encrypted transport layer, with authentication of the server; most browsers displays a little padlock when visiting a URL under HTTPS). This practice prevents the attacker from serving a page purporting to be from the legitimate `www.bankname.com`. However this doesn't do victims much good unless they make a point of checking that they are on the right website before entering their login credentials into it. Non-techies merrily click on links without knowing where they lead to and in any case they can't easily tell the difference between the legitimate `www.bankname.com` and `www.subdomain.bankname.com`

on one hand, and the malicious `www.bankname.otherdomain.com`, `www.bancname.com`, `www.hackeddomain.com/www.bankname.com` and so forth on the other.

From the viewpoint of the crook, the fact that some of the prospects will be able to tell that the email is fraudulent is not a big deal: they can easily compensate by sending out more emails. You don't care much about a low conversion rate if the per-user cost of prospecting is negligible. (These are the same economics we discussed with the porn/extortion fraud I mentioned in Section 1.3 on page 17.)

As for countermeasures, to the non-techies that asked me how they can protect themselves from phishing, I used to teach how to bookmark their bank's login page and to type their credentials into their banking site only when entering it through that privileged route[2]. As for myself, I use a more geeky solution that I don't think would work well for non-nerds: I simply refuse to give my regular email address to my bank—I give them a customised address that I don't use for correspondence with anyone else. That way, if I get an email purporting to be from the bank in my regular inbox, I am certain it is a fraud.

Nowadays, I use the browser's password manager and I never type a password into a website. If I visit a site where I should have an account, but the site asks me for a password and my browser doesn't know it, this is a strong hint that it's a phishing site rather than the genuine one.

A technical countermeasure that those in charge of the back-end can adopt is to enforce multi-factor authentication. This is effective against the phisherman because intercepting the other channels would be really difficult, at least if we talk about dragnet phishing as opposed to targeted spearphishing; but, of course, this comes at an additional usability cost for the user.

8.3 CSRF

A Cross-Site Request Forgery is an impersonation attack: the victim Victoria is logged into a website Walbank (say: an e-banking site) and the attacker Albert sends Walbank a malicious request (say: pay out some money) that Walbank honours as if it had been initiated by Victoria. Trouble is, the request really was, technically, initiated by Victoria because what Albert actually did was to trick Victoria into sending Walbank the malicious request he had crafted (rather than sending the request himself to Walbank directly).

Albert can achieve this by enticing Victoria to click on a hyperlink (in

[2]Of course this must be repeated for every website of value, so it is not an entirely effortless solution.

an email or on some other website) that contains the malicious request to Walbank. If Victoria was logged into her Walbank account (in some other browser tab, in the background) at the time of clicking the link, then Victoria's browser, on visiting the malicious link, would attach Victoria's cookies to it, making it appear to Walbank that the (malicious) request, which does genuinely come from Victoria's client, is supported by Victoria's credentials.

Note that this is somewhat similar to, but technically rather different from, a phishing attack. The similarity is that both attacks involve some element of social engineering, whereby Albert tricks Victoria into doing something without understanding what is actually going on. However, in phishing, the victim is tricked into performing the authentication protocol with the wrong website, whereas in CSRF the victim is "only" tricked into clicking on a URL without entering any of her credentials—an activity that might reasonably seem "safe". After all (victim's thought bubble), "I understand that entering my credentials might be risky, because I might give them to the wrong website; but if even just *clicking on links* is unsafe, what am I doing on the web in the first place?". In this sense, CSRF is more powerful than phishing, in that it will catch out even relatively alert users who might be able to spot basic phishing. But the attack is a bit harder to pull off.

The reason why the mere act of clicking on a link can be unsafe, even with an up-to-date browser whose known vulnerabilities are patched, is because Victoria is already logged into Walbank. Then the malicious link, which in this case isn't merely a request to view a page but a request to perform an action, is "executed" by the Walbank website using the credentials that Victoria already presented to Walbank when she first logged in.

How can a hyperlink also be an instruction or command to the website? Let's figure it out. Everyone is familiar with web forms, at least as a user: the web page does not merely contain text and images but also input fields that the user may fill in. Then there is a "Submit" button that, when clicked, sends the content of those fields to the website. How does this happen? When the "Submit" button is clicked, the browser sends the server one of two possible HTTP commands (depending on how the form was programmed[3]), called `GET` and `POST` respectively. In

[3]In theory, `GET` is intended to be used for idempotent queries (you ask the same question, you get the same answer), whereas `POST` is supposed to be used for requests that change the state server-side. Indeed, if you refresh a page, the browser will happily resend a `GET` request but should not repeat a `POST` request without confirmation from the user. However this is just a convention and there is no mechanism that enforces it: the choice of using one or the other is in the hands of the web developer who codes the page containing the form.

the case of `GET`, the parameters are encoded (as ampersand-separated name-equals-value pairs) in the URL that the client requests when the "Submit" button is pressed; whereas, in the case of `POST`, the parameters are encoded as part of the content of the HTTP request. The format of `POST` is a little more complicated to recreate manually and is more easily handled through a suitable library call.

Here are examples of the two methods. Imagine that the website `www.walbank.com` has a page with a form that allows you to pay money into another account. It displays two fields, `to` for the destination account and `amount` for the amount to be transferred (both integers, let's pretend), and a "Submit" button.

In the case of `GET`, the source of a bare-bones `pay-form-get.html` webpage might look as follows:

```
<html>
<body>
<form action="http://www.walbank.com/pay-action-get" method="get">
  Destination account:<input type="text" name="to"><br>
  Amount to be paid:<input type="text" name="amount"><br>
  <input type="submit" value="Pay">
</form>
</body>
</html>
```

Listing 8.1: `pay-form-get.html`

This might render as follows in the browser:

The `GET` request to send 5 GBP to account 12345678, sent by the browser to the website on submission of this form, might be as follows:

```
GET http://www.walbank.com/pay-action-get?to=12345678&amount=5
```

Conversely, in the case of `POST`, the source of the `pay-form-post.html` web page might be as follows, which is the same as the previous one except for the form's `method`:

```
<html>
<body>
<form action="http://www.walbank.com/pay-action-post" method="post">
  Destination account:<input type="text" name="to"><br>
  Amount to be paid:<input type="text" name="amount"><br>
  <input type="submit" value="Pay">
</form>
</body>
</html>
```

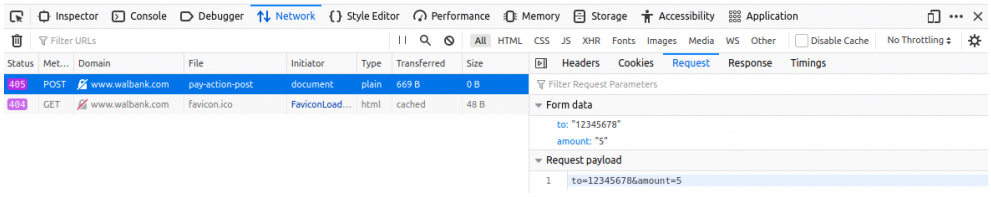
Listing 8.2: `pay-form-post.html`

This HTML code would render as follows, which looks indistinguishable from the previous one:

The `POST` request to send the same amount to the same payee, however, is somewhat different:

```
POST http://www.walbank.com/pay-action-post
```

The parameters are no longer encoded in the URL. They are instead in the "form data" as part of the body of the HTTP request. To see them, we must inspect the request with the tools mentioned in the SEED lab, or with a network analyzer such as Wireshark, or any other method you like. Here is Firefox's own "Web Developer > Network" built-in network analyzer tool showing the form data for this request:



Of course it's ultimately the same content but it is sent to the server in a different way. With a `POST` request, if you give the server just a URL, the parameters are missing because they are not encoded in it.

What does attacker Albert need to do in order to craft a URL that, if clicked by victim Victoria, will cause Victoria (provided she is logged into Walbank) to fill in the payment form with the parameters of Albert's

choosing, say Albert's account 66666666 and the amount of 5,000 GBP, without Victoria even realising she is submitting a form to Walbank?

The case of `GET` is relatively easy: Albert encodes his chosen parameters in the URL, yielding

```
GET http://www.walbank.com/pay-action-get?to=66666666&amount=5000
```

and all that is left for him to do is to persuade Victoria to click on that link while she is logged in. He may do so with a totally unrelated email ("click here for a video of cute funny cats") or with a post on social media, or even with a page on another unrelated website. Of course he has no guarantee that Victoria will click, nor that she will be logged into Walbank when/if she does. The two main levers that Albert has to increase his chances of success are to make his link more enticing to click on, and to target greater numbers of victims. In a pattern we have already witnessed with phishing and sextortion, even if the conversion rate is low, it costs him next to nothing to spam more people, and all that matters to him is the *product* of the three quantities: the probability that the victim clicks, the probability that the victim has an open session with Walbank at the time of clicking, and the number of victims that receive the link. This, to a first approximation, gives the number of attacks that will result in a transfer[4]. Note also, to increase the first number, the technique of embedding the URL as the `src` of a zero-by-zero pixels image in an HTML email: if the recipient's email client is configured to download images in incoming emails automatically, then receiving and opening the email will be sufficient to send the forged request, without the victim having to do anything else (though it is still the case that the request will only work if the victim was logged into the website at the time of opening the email).

But it is not up to Albert to choose whether to use `GET` or `POST`: he must match whatever method is actually in use at Walbank. In the case of `POST`, he must first capture, observe and dissect a genuine `POST` request sent through Walbank's form; and then recreate a form on his own attacker website that generates a `POST` request in the same format, except with his own chosen parameters (you will be doing that in the SEED lab). And then he must get Victoria to submit his form. However, while he might plausibly get away with persuading Victoria to click on a

[4]Of course this is only a toy example to illustrate the technical issues around CSRF but is not a realistic depiction of actual scams. For that, in order to make the attacks actually succeed, Albert would have to take into account the fact that the bank would trace him as the owner of 66666666, the fact that the bank might be able and willing to reverse payments (particularly ones within accounts at the same bank), and hence the necessity of using unwary "money mules" as intermediaries and so on.

link, it would be a bit harder for him to persuade Victoria to submit a *form* from some random website[5].

What Albert the attacker can do, however, is to craft a page (on his own website) that contains a *hidden* form and also a JavaScript callback that automatically submits the form as soon as the page is loaded. If he sets things up that way, then the simple fact of visiting his URL will be enough to send the `POST` form, and he will be in as good a situation as with `GET`, from the viewpoint of his task of persuading Victoria.

The following should be obvious by now but let's say it explicitly one more time anyway: the CSRF vulnerability is a form of impersonation, but it requires the victim (prover) to be already logged into the website (verifier), which technically means that the website (the verifier) has deposited some login cookies into the victim's browser (the prover), and that these cookies will be sent back to the website when the browser issues subsequent requests to the same site. The impersonator does not interact directly with the verifier: instead, he causes the victim to interact with the verifier, using her own genuine credentials (those login cookies), but requesting an action defined by the verifier. The attacker builds a request and, through social engineering, tricks the victim into clicking on a link: this action sends the malicious request to the website where the victim has an account (together with the cookies), without her being aware that she did so. The attacker does not steal the cookies: he never even *sees* them. He just gets the victim to present them to the website herself, unsuspectingly validating the attacker's malicious request.

8.4 Countermeasures

Anyone who suggests that, to counter CSRF, we should tell users not to click on dubious links, should fail the test as a security professional. Such advice is unrealistic and counterproductive. The whole point of the Web being such a powerful and revolutionary tool for disseminating ideas is the paradigm of clicking on links to sites you don't know. It is up to us as security engineers to make this activity safe, somehow sandboxing the activities of the browser so that clicking on a link can not, by itself, cause malicious side effects.

The sensible way to counter the CSRF attack is to let the web server distinguish whether the request is cross-site or not. Ideally, the web server should only honour requests that originated from pages on its own site, and should reject requests coming from elsewhere. But it can't tell

[5]Although nowadays, with every website popping up a form to request your consent to the use of cookies, it might not be that hard after all...

the difference, because of the statelessness of HTTP. On the client side, however, the browser *can* tell the difference: when the browser sends the `GET` or `POST` request to Walbank, it knows very well whether or not it came from pressing the "Submit" button on an HTML form it had received from Walbank. So it should be the job of the browser to let the server know. Of course people can do what they like with their browser, including rewriting it so that it won't follow the rules; but here the browser in question is that of the victim, not of the attacker, and the victim has no incentive to make her browser misbehave. Thus, provided we can come up with a working scheme, the victim could protect herself by using a browser that complies with this scheme (of telling the server whether the request the browser just sent was cross-site or not) and it should not matter that the *attacker* might use a browser that did not comply with the scheme.

Section 12.5 of your W. Du, *Computer Security* 3rd ed textbook briefly describes three such schemes for telling the web server whether the request was cross-site or not: the `referer` header, the `SameSite` cookie attribute and the secret token.

8.4.1 `referer` header

In the first of these schemes, the browser includes the `referer` header in the HTTP request, mentioning the URL of the page that the user was on when clicking the link. The trouble with this approach is that the `referer` header is optional, and is often disabled out of privacy concerns. Disabling the header causes false positives, i.e. requests classified as cross-site (for lack of a `referer` header stating that they came from Walbank) even though they weren't.

8.4.2 `SameSite` cookie attribute

The second scheme, the `SameSite` cookie attribute, was introduced by Google Chrome in 2016 but its adoption and deployment (both by Google and by other browser vendors, but more importantly by websites) has so far only been partial because of backward compatibility concerns. A cookie marked as `SameSite=strict` is only sent back to the site if the site of the cookie is the same as the site that the browser is currently visiting.

In other words, if you were logged into GitHub, and your authentication cookie were marked `SameSite=strict`, and a GitHub page you visited had a link to another GitHub page, then, on clicking that link, your authentication cookie would be sent along, and you would be able

to open the second page seamlessly. If instead you received a message in Gmail containing a link to the same GitHub page, if you clicked that link, your GitHub autentication cookie would *not* be sent along and thus you would not be able to access the page, despite being already logged into GitHub in that same browser. This loss of functionality as the price for the extra security is the reason why web developers have been reluctant to adopt this countermeasure (although it would stop CSRF), and why the lower-security settings of `SameSite=Lax` and `SameSite=None` have also been introduced as alternative trade-offs that offer less protection but more convenience.

Taking the big picture view, this kind of analysis is at the core of systems security. A website may prefer to keep its customers less secure provided it retains them as customers, as opposed to offering greater security but losing a proportion of its customers to a competitor that puts user convenience ahead of strict security. (And that's before even talking about the cynical truth that "if you're not paying, you're not the customer; you're the product". Thus user security might matter even less to the website operator, because those whose security is diminished are not even his customers.)

8.4.3 Secret token

The third scheme we discuss, instead, is one that is implemented at the web application level (server-side rather than client-side) and therefore it can be deployed immediately, without having to wait for all major browsers to adopt some common standard about the matter, as was the case for `referer` and `SameSite`. The advantage is that a website that wants to protect itself against CSRF may adopt this scheme unilaterally, without having to wait on any other party to do anything. The complementary downside is of course that the scheme has to be reimplemented in every web application.

The web application used as a target in the SEED lab, called Elgg, uses this third scheme. In the SEED lab exercise, the countermeasure has been pre-disabled for you, to allow you to experiment with the CSRF attacks of tasks 2 and 3. You re-enable it in task 4.

The idea of the secret token scheme is that the website, whenever it serves a form, computes a new secret[6] and embeds it in the page. The form contains a hidden field that must be filled with this secret token. When the browser submits the form through a `GET` or `POST` request, the token is automatically sent along, because the HTML that generates the

[6]For example a Message Authentication Code (see footnote 1 on page 108) of the timestamp and the user session ID, keyed by the website's MAC key.

request contains the token as an embedded constant. The server checks the validity of the token before accepting the submitted form.

The attacker is not able to incorporate a valid token into his forged request: even if he uses his own browser to visit the website and get a genuine token, the token will be linked to the attacker's session ID, not the victim's. Therefore the attacker will not be able to incorporate it in the fraudulent `GET` or `POST` request that he will trick the victim to submit on his behalf. Even if the attacker were able to guess (or attempted to brute-force) the session ID of the victim, he still would not be able to generate a valid token without knowledge of the website's MAC key.

Why wouldn't the secret token be sent back to the server by the victim's browser together with the victim's login cookie? Because it is not held in a cookie. It is instead generated afresh by the server before serving the page, then embedded by the server into the page. The secret token *for Victoria* is not even created until the page is served *to Victoria* and, in the typical CSRF attack, Victoria hasn't even been served that page. She is just logged into the Walbank website, visiting some other page, when Albert persuades her to click on a link that simulates her sending a `GET` or `POST` request coming from the form page, but actually crafted by Albert. Albert has no way to copy Victoria's token from the genuine form page, first because the page might not even have been served to Victoria in the first place and second because, even if it had been, the Same Origin Policy[7] would prevent the fake page on Albert's website (even if opened by Victoria's browser that has the genuine page opened in another tab) from reading the token from the original page.

The timestamp is also sent along back and forth, separately, in plaintext, via another hidden field or Javascript variable. This allows the server, on verification, to re-compute exactly the same secret and also to check that the timestamp is fresh, to prevent replay attacks.

[7] The Same Origin Policy, enforced by standards-compliant web browsers, restricts scripts on a given web page from accessing data in other web pages. Only data in pages with the same origin (URI scheme, host name and port number) may be accessed.

Chapter 9

Cross-Site Scripting

Textbook

Study Chapter 13 in W. Du, *Computer Security* 3rd ed.

SEED labs

Complete the following lab: Web Security | Cross-Site Scripting Attack Lab (Elgg) (tasks 1 to 6; task 7 optional but recommended).

It is a good idea to read the last section ("Guidelines") of the online briefing document for this SEED lab before starting to work on the tasks.

9.1 XSS

Cross-site scripting[1] is, like CSRF, another kind of impersonation attack in which the attacker causes a malicious action to be executed while the victim's cookies are active. The attacker Albert tricks the victim Victoria into executing some malicious code (typically JavaScript) in her browser. Since the malicious code executes in Victoria's browser with Victoria's credentials, it can do most of the things that Victoria has authority to do.

Some people confuse CSRF and XSS. They are somewhat similar: both involve some modest amount of social engineering (tricking the victim to visit some URL) and both exploit the victim's authentication

[1] Abbreviated as XSS rather than the more obvious CSS to avoid confusion with Cascading Style Sheets.

cookies for impersonation, requiring the victim to be logged in. But CSRF only lets the attacker run, as the victim, one of the possible transactions already offered by the website (submitting a form—the website must be expecting responses to that form); whereas XSS lets the attacker run arbitrary code, as the victim, in the victim's browser, which gives the attacker greater freedom: Albert the attacker can do his own *scripting*, rather than merely a *request forgery*.

One must first question the basic assumptions here. If people are willing to turn on JavaScript, they are already willing to accept that arbitrary websites can execute arbitrary code in their browser, right? So what do they complain about if some of that code is malicious? Shouldn't they just disable JavaScript? Well, in today's world that battle has largely been lost: even though some of us refused for many years to run with JavaScript enabled, nowadays some 97% of websites rely on JavaScript[2] and basically nothing works properly if you leave it off. So, what mitigations are in place? Well, there is some amount of sandboxing, with the intention that the JavaScript you get from one web page can only access objects in another web page if both pages have the same origin[3]—the so-called Same-Origin Policy[4] we mentioned in the previous chapter (footnote 7 on page 118). And of course the JavaScript served to your browser in a page from Walbank can do anything it wants to your Walbank account—but that's not an issue because code under control of Walbank could do anything it wanted to your Walbank account *anyway*, by doing it at the server side even if JavaScript didn't exist. The point of the Same-Origin Policy is that the code from Walbank should be allowed to do whatever it wants to Walbank objects, but not for that should it be allowed to mess around with any of the pages you have open concurrently in other browser tabs—for example reading the messages in your webmail.

The core idea of the XSS attack is to violate the assumption that the JavaScript you get from Walbank was written by Walbank—and then to do devious things to your Walbank account while logged in as you at Walbank. Attacker Albert arranges for victim Victoria to receive malicious JavaScript written by him, but which is served to her by Walbank, and which Victoria therefore trusts while she is on Walbank's website.

[2] Actually 97.9%, according to `w3techs.com` in May 2022: https://w3techs.com/technologies/details/cp-javascript.

[3] Technically defined as the combination of URI scheme (e.g. HTTP vs HTTPS), host name and port number.

[4] There is a tension here between security (disallowing cross-site interactions because the site doing the manipulation could be malicious) and functionality, whereby websites come up with new and exciting (lucrative?) ways that different websites could interact and lobby browser vendors to allow such behaviour (easier for websites that invested into making and distributing their own browser).

In a sense, this is the reverse of CSRF: there, Albert fooled the *server*, Walbank, into believing that the fake request originated from the authenticated client, Victoria. In XSS, instead, Albert fools the *client*, Victoria, into believing that the fake code originated from the server where she has an account, Walbank.

9.1.1 Non-persistent (reflected) XSS attack

What Albert may do is to exploit situations where Walbank is serving Victoria a page that isn't merely static but which contains content that depends on user input in some way or other. And then Albert needs to manipulate the user input so that the page generated by Walbank and served to Victoria contains the malicious JavaScript payload that Albert wants to execute within Victoria's browser.

There is another issue in all this, which is one that frequently recurs in this course: *malicious code disguised as data*. Normally, when Walbank serves a page with content that is a function of user input, the expectation is that the user input that is being reflected in the Walbank-generated page is "just data", not *executable* content. But many websites don't check, or don't check carefully enough (and even when they check it's really hard to be absolutely sure). As a result, some executable content may slip through. Let's first assume that Walbank does not perform any sanity checks[5] and see what Albert can do to cause his executable JavaScript to be served to Victoria when she visits the Walbank site.

A **non-persistent XSS** attack exploits the case where some page of the website has a form accepting user input and where the response to that form reproduces that input. Imagine that Walbank embeds a search engine in a corner of its pages: you type `foreign exchange` in the search box and Walbank returns a page saying "here are some pages I found that relate to `foreign exchange`", followed of course by links to those other pages. The important point here is that Walbank *repeated your text verbatim* back to you. Now, what if your text contained some HTML tags, such as `foreign <b>exchange</b>` to make the second word boldface? Would the result page again reproduce your text byte for byte, tags included, causing the tags to be interpreted by your browser and making that word bold? Or would it somehow quote your text before reproducing it on its output page, transforming `foreign <b>exchange</b>` into `foreign &lt;b&gt;exchange&lt;/b&gt;` causing the angle brackets to be displayed rather than parsed as tag delimiters? Whether it would do one or the other is a matter of policy and implementation; but, if

[5] Because, in many real-world cases, no checks are performed; and, even when there are checks, there may still be false negatives.

the website is allowing the boldface `<b>` HTML tags to be interpreted and rendered by the browser, then it might possibly also let through an HTML tag such as `<script>` which, when interpreted by the browser at the time of rendering the response page, would cause a script to be evaluated as code, and thus *executed* in the viewer's browser. Albert's attack therefore consists of crafting an input for that form that will cause the desired malicious script to be executed when the response page is rendered; and then to arrange for Victoria to submit that input to the form. This in turn could be done by preparing a `GET` or `POST` URL that did the submission and then tricking Victoria into clicking on it, using similar social engineering techniques to those mentioned in the context of CSRF in chapter 8 on page 107.

9.1.2 Persistent (stored) XSS attack

In a **persistent XSS** attack, instead, Albert covertly stores his malicious executable at Walbank, server-side. He exploits a situation whereby the website allows users to post content that will be seen by other users, such as messages on a bulletin board or discussion forum, or comments on a blog post, or profile pages on a social media site. Provided that HTML tags are not sanitised, this lets Albert store his executable payload in the Walbank back-end, on a page that Walbank will happily serve to other users. If Victoria (or any other Walbank user) is logged into her Walbank account while she views the page, Albert's malicious code (blessed by the legitimacy of having been served by Walbank) will execute in her browser with her Walbank cookies active, and will therefore be able to do most of what she could do herself in that context.

The privilege escalation, in this web context, is no longer "gaining root access", as it was in the attacks on setuid programs we saw in Chapter 3 on page 41, but rather "getting the privileges of the victim". Once the attacker can run arbitrary JavaScript code in the victim's browser while the victim is logged in, he can edit the victim's profile, change (deface) any pages that the victim has write access to, post messages purporting to be from the victim and, in summary, perform any actions that the website allows the victim to perform: transfer funds, buy items...

For any juicy (confidential) piece of information that is available on the victim's pages (her account balance, her credit card number, even her login cookie), it is trivial for the attacker to display it on screen by popping up an alert box:

```
<script>alert(document.cookie);</script>
```

Trouble is, for the attacker, that the JavaScript is executing in the

victim's browser, so it's on the victim's screen that the alert box is displayed. How can the attacker exfiltrate any of the information that his malicious JavaScript can access and manipulate? (This is one of the initial tasks you will have to perform in the SEED lab.) An ingenious idea is to use "input as output". There are no restrictions on JavaScript requesting input from other websites, for example by adding HTML `<img>` tags to the document that would load pictures into the page from other sites; but it takes a bit of lateral thinking to realise that this input facility is in fact also an output facility because it is also sending something in the opposite direction—the name of the picture file being requested! So, from the malicious JavaScript now running in Victoria's browser, Albert simply requests non-existent images from his own server, where the names of the images encode the information that he wants to exfiltrate—for example, Victoria's login cookies. He then checks the logs of his web server to see what requests came in.

Another one of the tasks you will be doing in your SEED lab, described in detail in your W. Du, *Computer Security* 3rd ed textbook, replicates the historic XSS worm that hacker Samy Kamkar (1985–) unleashed on the social networking website MySpace in 2005. The payload of the worm was simply to add Samy as a friend and to add a declaration to that effect in the victim's profile: "but most of all, samy is my hero". The worm was self-replicating, so any other user viewing an infected profile would execute that same payload and in turn become infected. When it came out, the Samy worm was the fastest-spreading piece of malware ever seen: within less than a day from its release it had infected over a million user profiles. There are several instructive write-ups about this incident on the net, some more narrative and others more geeky. A relatively short one with a wealth of technical details is the one on Kamkar's own website, which also contains the full code of the worm: https://samy.pl/myspace/tech.html.

9.2 The attacker's toolkit

To mount a persistent XSS attack, Albert must first of all find a vulnerable page at Walbank—one that lets him store his HTML input without filtering out his JavaScript. Once he has that foothold, to craft a payload that does anything interesting as Victoria, he will generally want to *submit a form* to Walbank as Victoria. As we saw with CSRF, Albert must comply with whatever kind of form has been defined by Walbank. The case of `GET` is easily handled because both the destination and the parameters are encoded in the URL. The case of `POST` is somewhat more elaborate but this is not a problem since Albert is in a position to write

his own JavaScript. He builds an `XMLHttpRequest` object, populates it appropriately (including the parameters of the request in the content of the request) and then invokes its `send()` method. The parameters may include the security token that Walbank requires as protection against CSRF (cfr. Section 8.4.3 on page 117): this time Albert has no trouble including it in his request because it is available as a variable to the JavaScript code running on that Walbank page.

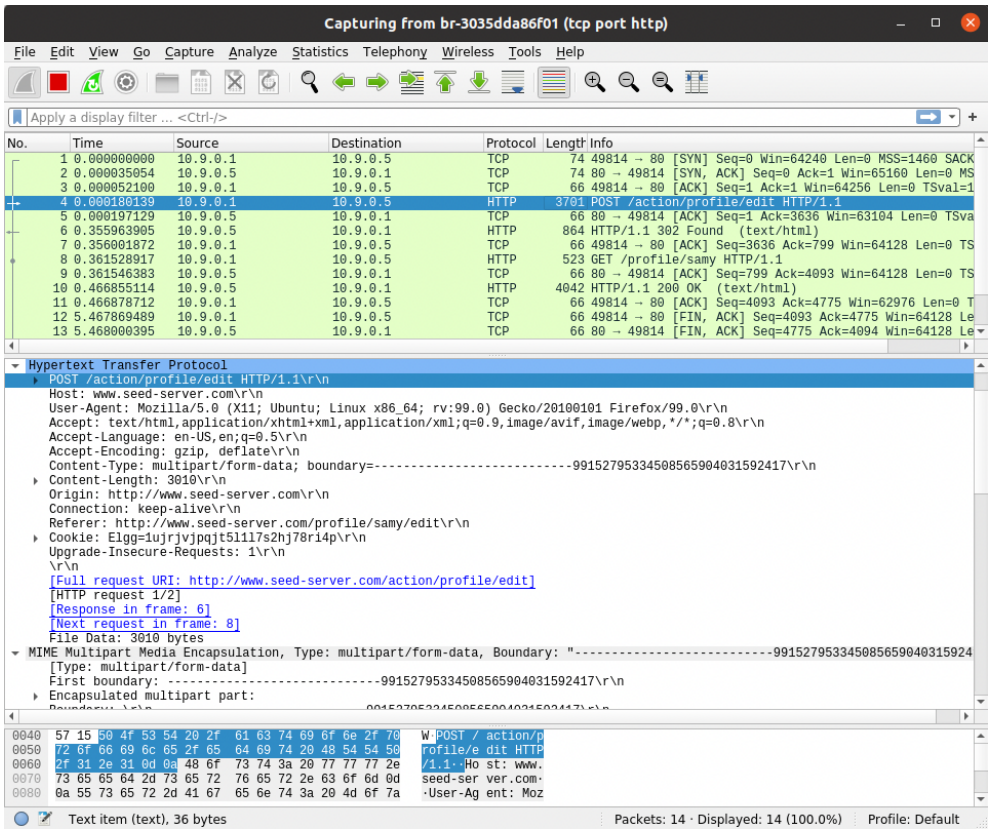
You will learn how to do this in the SEED lab, which provides boilerplate JavaScript code with blanks for you to fill in once you work out what the actual request ought to be. Here is a fragment of the JavaScript boilerplate that submits a `POST` form. You have to figure out, among other things, the `content` and the `sendurl` in lines 5 and 6 respectively.

```
1  <script type="text/javascript">
2      window.onload = function(){
3          var timestamp="&__elgg_ts="+elgg.security.token.__elgg_ts;
4          var token="&__elgg_token="+elgg.security.token.__elgg_token;
5          var content=...;
6          var sendurl=...;
7
8          var Ajax=new XMLHttpRequest();
9          Ajax.open("POST", sendurl, true);
10         Ajax.setRequestHeader(
11             "Content-Type", "application/x-www-form-urlencoded");
12         Ajax.send(content);
13     }
14     </script>
```

Listing 9.1:

And how *do* you work out the correct parameters? By observing a genuine transaction as it goes through, for example with Wireshark or with Firefox's built-in "Web developer tools / Network". You'll be doing that as well in the SEED lab. You'll pick up that there are many ways of encoding the content and that, while not all of them are appropriate for all types of content, many things can be encoded in more than one way. Here is part of what Wireshark shows[6] when Samy the attacker changes his own profile to something innocuous, on his way to building his payload.

[6] You'll have to figure out what traffic filter to set (hint: HTTP) and what interface to listen to (hint: bridge). You will need to have configured your VM to allow the network adapter to work in promiscuous mode (meaning: listen to all traffic, not just that aimed at your own address), as explained in the SEED lab setup instructions, and you may need more privileges than a regular user in order for Wireshark to have raw access to the network adapter.



Note that the transaction you observe might encode the form data in a relatively complicated way, such as the

```
Content-Type:  multipart/form-data
```

that we see here in the Wireshark capture, but that you are free to submit your own response using a different encoding, such as the easier-to-write

```
Content-type:  application/x-www-form-urlencoded
```

of listing 9.1 on page 124 which is in fact similar to the style of `GET` parameters. Look these things up on the Web as needed—you are not expected to know everything already, but you *are* expected to acquire the valuable skill of solving practical problems by learning the missing bits as you go along on an as-needed basis.

9.3 Countermeasures

As in other contexts involving data-as-code attacks, there are two broad approaches to preventing XSS. The first is sanitisation, which in turn can

be done in at least two ways: either by filtering out any executable code from the user input; or, alternatively, by filtering out the metacharacters and tags that cause the attacker input to be parsed as code, leaving the JavaScript source in the input but just as text. Sanitisation can be a useful first line of defense, and is certainly better than nothing, but it is generally difficult for it to cover all cases.

As proof of that, the Kamkar write-up mentioned above starts as follows:

> Myspace blocks a lot of tags. In fact, they only seem to allow `<a>`, `<img>`s, and `<div>`s...maybe a few others (`<embed>`'s, I think). They wouldn't allow `<script>`s, `<body>`s, `onClick`s, `onAnything`s, `href`'s with javascript, etc... However, some browsers (IE, some versions of Safari, others) allow javascript within CSS tags. We needed javascript to get any of this to even work.
>
> Example: `<div style="background:url('javascript:alert(1)')">`

The fact that Kamkar was eventually able to circumvent sanitisation should not be ascribed solely to MySpace's incompetence: it has repeatedly proved very hard to anticipate all the possible ways through which attackers may slip under the radar of this kind of checks. If adopting input sanitisation, it is a good idea not to roll one's own filter but to rely on a well-tested library developed and maintained by the developers of the web framework in use.

But there is an alternative to sanitisation that is somewhat more principled. The root cause of the XSS vulnerability is that the browser does not know the provenance of the JavaScript code it is being asked to execute. Was it written by Walbank or by Albert? If we could give Victoria's browser that piece of information, we could also instruct it to execute only the code that was genuinely written by Walbank[7].

As we have seen, in XSS, Albert's code is *served* from Walbank's server—indeed it is embedded in a page served by Walbank. So it seems as if that battle is already lost. But what if HTML code (which is not executable) were not allowed to embed JavaScript code directly, and were instead only allowed to embed URL references to JavaScript source code files to be downloaded? Then Victoria's browser would be in a position to tell where the JavaScript came from, and it would be able to reject that not coming from Walbank. Whether it did reject it or

[7]Recall that Walbank is already, by necessity, within Victoria's trusted computing base, because it could do anything it wanted to Victoria's assets (server-side) anyway, regardless of any protections that Victoria might activate in her browser.

not would be decided by a server-specific Content Security Policy (CSP), specified by the website in the HTTP response headers of the pages it served. As usual, from the security viewpoint of preventing XSS, it would appear preferable to exclude embedded JavaScript altogether and force all JavaScript to be imported from URLs; but there is the by-now-familiar trade-off with developer convenience and backwards compatibility.

The optional task 7 of this SEED lab gives you the opportunity to experiment with CSP in various forms to block XSS attacks.

Chapter 10

Human factors

It is only relatively recently (over the past couple of decades or so) that the security research community has come to acknowledge the importance of human factors in cybersecurity. Our purpose as security engineers must be not merely to secure computers but to secure *systems* that depend on computers; and these complex socio-technical systems include, besides the computers and their peripherals and their software and their network connections to other computers, also the people who use those computers, the people who maintain them, the people who write software for them, the people who define the security policies for them and so forth. All of these elements are interconnected. And, while people who only deal with the organisational, sociological and relationships level but lack the technical competence to write the sequence of bytes to exploit a buffer overflow vulnerability cannot in my opinion be considered cybersecurity experts, conversely I also believe that a hardened techie who is highly skilled at assembly language and network protocols but lacks sufficient understanding of how non-geeks use computers will *never* be able to design and build truly secure systems. A strong security expert needs both skill sets.

A holistic approach is needed. The most valuable, most sought-after and most highly paid security expert is the one who can solve real problems. It's someone who is simultaneously competent about the technical aspects but also able to relate to the role of computer systems in serving and supporting the inhabitants of a digital society that is populated primarily by non-nerds—because it is for them, ultimately, that the systems must work well.

As I said about other subtopics, a whole lecture course could be devoted *just* to human factors in security. Whatever I might choose to say in this chapter will omit many other important things. So let me arbitrarily select just three topics, with the intention of arousing your interest and curiosity. There is plenty more interesting research to be

done in this direction: what would *you* like to contribute to the field if you one day continued into graduate studies?

10.1 Users are not the enemy

If the technical intricacies of computer security are too obscure for ordinary software developers (who continue to write code containing buffer overflow and SQL injection vulnerabilities), imagine for non-geek users! There is a wide mismatch between security techies and regular human beings, both in their respective understanding of how computers and their security mechanisms work on one hand, and in how much they care about those security mechanisms on the other hand. This has too often led to security techies adopting a detrimental us-and-them attitude of "the system I designed would be totally secure *if only* those !#%$% users behaved as instructed", and you can hear the contempt in the often-repeated assertion that "the human is the weakest link", which carries the silent subtitle of "computers are infallible, it's just those stupid humans who make mistakes". This misguided mindset only serves to alienate users, who in turn see the security techies as arrogant gatekeepers who put annoying obstacles in their way (think VPNs, multi-factor authentication tokens and so forth), which they need to endure (or somehow bypass) in order to get any useful work done. These attitudes put the security techies and the non-techies at loggerheads—a highly counter-productive setting, whereas the two groups should instead cooperate to protect their organisation from attackers[1]. Highlighting the fact that the infight between security administrators and users was an absurd waste of resources for the organisation was the thesis of Adams and Sasse[2] who, in 1999, published one of the first articles in the security usability literature—still a recommended read today. Valuable insights from that work include the following:

- Insufficient communication with users produces unusable systems: if the security designers are techies, and they don't talk with regular users during development, the end result will only be usable by other techies[3].

[1] Whether external or internal—insider fraud is a thing too. But that's a separate story.

[2] Anne Adams and Martina Angela Sasse. "Users are not the enemy". *Commun. ACM* 42, 12 (Dec. 1999), 40–46. https://doi.org/10.1145/322796.322806.

[3] Obvious examples of systems that non-geeks find difficult or impossible to use include the traditional Unix command line utilities and the pioneering public key encryption program PGP—or, possibly even worse, its well-intentioned free-software replacement GPG.

- Users forced to comply with security mechanisms incompatible with their work practices will necessarily look for workarounds[4]. Users are not *bad* people: they just need to get their work done, because that's what determines their remuneration and promotion—not whether they complied with the security policy.
- The claim that users are never motivated to behave securely is wrong: treat them as stakeholders and they will gladly cooperate. Provide feedback, guidance, awareness; and *usable* security mechanisms.

Another valuable insight about the dynamics of this interaction between security administrators and regular employees comes from a later paper from the same research group[5]. The thesis of that paper, based on interviews with corporate employees on whether they complied with their company's security policy, is that users have a finite supply of goodwill, which the authors indicate as a "compliance budget". Every act of complying with some annoying bit of corporate security policy makes a withdrawal on that budget. When the balance is zero or negative, there is no goodwill left and the user will resort to workarounds rather than complying with further requests, by then perceived as totally unreasonable. The paper thus recommends managing the compliance budget as wisely as one would any financial budget, spending its finite resources only on what matters most. In other words, harassing users with petty security requests might have the counterproductive result that they won't follow the security policy on more serious matters.

> "Once the compliance threshold is crossed, security effectiveness drops sharply as altruistic behaviour is increasingly exchanged for selfish options."

10.2 Prospect Theory: an analysis of decision under risk

Psychologists Amos Tversky (1937–1996) and Daniel Kahneman (1934–2024) invented **Prospect Theory**[6] in 1979 and contributed to the establishment of the discipline of behavioural economics. Their work led

[4]Such as sharing passwords to delegate access rights, or transferring files through USB drives rather than via the recommended encrypted channels.

[5]Adam Beautement, Angela Sasse and Mike Wonham, "The compliance budget: managing security behaviour in organisations", *Proc. NSPW 2008*. http://www.nspw.org/papers/2008/nspw2008-beautement.pdf

[6]Daniel Kahneman and Amos Tversky. "Prospect Theory: An Analysis of Decision under Risk". *Econometrica* Vol. 47, No. 2 (Mar., 1979), pp. 263-292. DOI:

to a Nobel prize in Economics for Kahneman[7]. Kahneman's fascinating Nobel lecture, whose video[8] and write-up[9] are available online on the Nobel Prize website, is a good investment of an hour of your time. It is worth understanding at least the basic ideas of prospect theory because of the insights they offer on the decision strategies that people use in security-relevant contexts.

The previous mainstream theory of how people take financial decisions, called Expected Utility Theory, had been put forward by Daniel Bernoulli (1700–1782) in 1738. It explained why the first million you earn is worth more to you than your second, and why people seem risk-averse. For example, would you accept the following gamble?

- Heads, you win 101 $
- Tails, you lose 100 $

Clearly, assuming a fair coin, you stand to gain a little more than you'd lose. Basic arithmetic tells us that the expected outcome in terms of wealth is

$$+101\ \$ \cdot 50\% - 100\ \$ \cdot 50\% = 0.5\ \$$$

and therefore, since it's positive, you should have a very slight incentive to gamble. However, in practice, most people refuse to risk their 100 $ unless the incentive is rather more substantial. Why? Are they being rational? What would *you* do?

The value you ascribe to money, Bernoulli said, does not grow proportionally to how much of it you have. If you have wealth w on the x axis, and the utility of wealth $u(w)$ on the y axis, the graph of $u(w)$ is not linear but logarithmic: the more money you have, the less each extra dollar is worth to you (makes sense, right?). If you perceived a certain increase in utility when your wealth went from 10,000 $ to 100,000 $, then, to perceive the same increase in utility again, you'd need to get not merely to 190,000 $ (*adding* the same amount of 90,000 $) but to a million dollars (*multiplying* by the same factor of 10×).

Bernoulli then explained that the rational strategy is to base one's choice on the expected outcome in terms of *utility of wealth*, rather than in terms of wealth. From this viewpoint, because of the concavity of

10.2307/1914185. https://web.mit.edu/curhan/www/docs/Articles/15341_Readings/Behavioral_Decision_Theory/Kahneman_Tversky_1979_Prospect_theory.pdf.

[7] Tversky would have likely been a joint recipient if he had still been alive in 2002.

[8] https://www.nobelprize.org/prizes/economic-sciences/2002/kahneman/lecture/

[9] https://www.nobelprize.org/uploads/2018/06/kahnemann-lecture.pdf

$u(w)$, the utility of being richer by 101 \$ is closer to the utility of your current wealth than the utility of being poorer by 100 \$. In other words, given that $u(w)$ is concave, the weighted average of the two outcomes, although it is slightly positive in terms of wealth, is slightly negative in terms of utility! This was a powerful insight.

However, a couple of centuries later, Tversky and Kahneman produced experimental evidence of specific cases in which this Expected Utility Theory was wrong, in the sense that its predictions consistently diverged from the actual choices of the experimental subjects.

In Tversky and Kahneman's experiments, while subjects were indeed risk-averse some of the time, for the reason explained by Bernoulli's Expected Utility Theory, in other cases they were risk-seeking. Why? Attempt the following experiment yourself, while trying your best to avoid the implicit bias that might come from what I already told you till now.

> Imagine that you face the following pair of concurrent decisions. First examine both decisions, then indicate the options you prefer.
>
> **Decision (i).** Choose between:
>
> **A.** A sure gain of \$240
>
> **B.** 25% chance to gain \$1000, and 75% chance to gain nothing
>
> **Decision (ii).** Choose between:
>
> **C.** A sure loss of \$750
>
> **D.** 75% chance to lose \$1000, and 25% chance to lose nothing

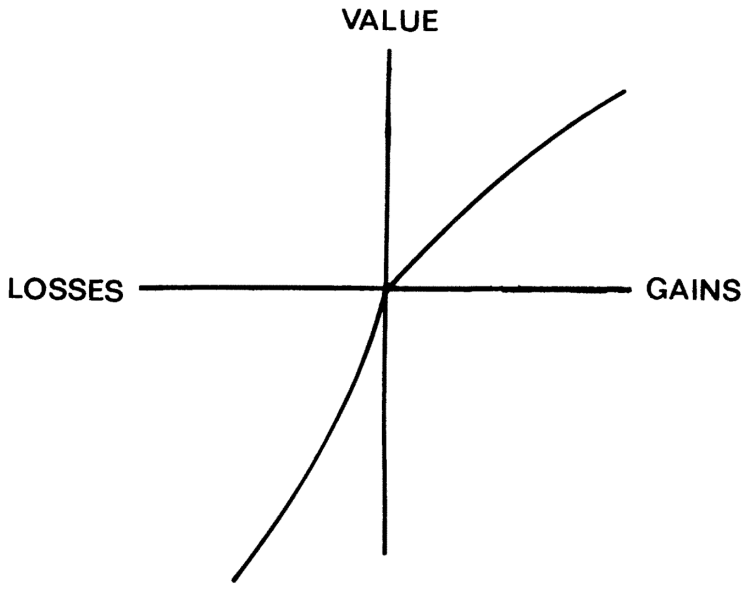
What pair of choices did you pick? (A, C), (A, D), (B, C) or (B, D)? I know, the eye is quick and the flesh is weak, but for a greater understanding of this material, please STOP, re-read the experiment and commit to your own choices *before* reading on.

Tversky and Kahneman administered the experiment to 150 subjects, as reported in their 1981 paper[10], and found that, in the case of Decision (i), an overwhelming majority (84%) chose the certain gain of option A versus the uncertain gain of option B—even though the expected gain in option B is 250 \$, marginally greater than that of option A. This risk-aversion is exactly as predicted by Expected Utility Theory. However,

[10] Amos Tversky and Daniel Kahneman. "The Framing of Decisions and the Psychology of Choice", *Science*, 211(4481):453–458, 1981. https://doi.org/10.1126/science.7455683.

in the case of Decision (ii), an equally overwhelming majority of the experimental subjects (87%) preferred the gamble of option D, which included the possibility of losing nothing, versus the certain loss of option C, even though the expected loss of wealth in option D was the same as in option C and therefore, by Expected Utility Theory, the expected loss in terms of utility would have been greater (worse) in option D than C.

The discriminant between risk-aversion and risk-seeking, they explained, was whether the problem was framed as a gain or a loss. In other words, although people were indeed unwilling to gamble in order to secure a gain, they preferred to gamble in order to avoid a loss. Another insightful finding was that people were significantly more sensitive to losses than to gains: they disliked a loss *much more* than they liked a gain of the same magnitude.



On that basis, the qualitative graph illustrating "a hypothetical value function" in Tversky and Kahneman's original 1979 prospect theory paper, repeated identically in their 1981 paper, has the following features:

- The x axis is not "absolute wealth", as it was with Bernoulli's Expected Utility Theory, but rather "relative changes in wealth with respect to a reference position" (the origin). This accounts for the influence of "framing effects".

- In the first quadrant (positive increase in wealth and positive increase in utility), the curve is concave, which accounts for risk aversion with respect to gains.

- In the third quadrant (decrease in wealth leading to decrease in utility), the curve is instead convex. This change in concavity accounts for risk-seeking behaviour with respect to losses.

- The curve is much steeper (by a factor of about 2) in the third quadrant than it is in the first. This takes into account the fact that people are much more sensitive to losses than to gains. If an increase in wealth results in a certain increase in utility, a decrease in wealth by the same amount results in a much greater (in absolute value) decrease in utility.

- Both the positive and the negative portions of the graph flatten out towards the horizontal: both higher gains and higher losses become progressively less relevant in proportion. The second million you earn (or lose) does not matter as much as the first.

An understanding of the psychology of decision-making, including various cognitive biases that may lead to outcomes that contradict what a "logical" or "rational" approach might otherwise suggest, is important to the security engineer. We need to design systems that will be secure in the face of how real-world users will actually use them. Towards that, the more we understand how users will respond and the better equipped we will be to build a system that nudges users towards a safer outcome.

10.3 Understanding scam victims

My coauthor for this work[11] was one of the most interesting and colourful characters I ever collaborated with on an academic paper: magician, deception expert and TV presenter Paul Wilson.

Check out this EFF-reported Facebook phishing attack targeting Syrian activists in 2012: the page looks like Facebook but was set up by pro-Syrian-government hackers to entrap Syrian activists.

[11] Frank Stajano and Paul Wilson. "Understanding scam victims: seven principles for systems security." *Commun. ACM* 54, 3 (March 2011), 70–75. DOI:10.1145/1897852.1897872. https://www.cl.cam.ac.uk/~fms27/papers/2011-StajanoWil-scam.pdf.

Security engineers build system defenses largely based on how they think users should respond to threats. From the engineer's viewpoint, just reading the URL would tell you straight away that this isn't Facebook. But normal users are not engineers: they respond differently. They don't understand the syntax of URLs, nor the difference between the HTTPS padlock and a picture of a padlock in the page itself. As a result of this mismatch, systems are vulnerable.

We need to understand how users really behave, and what psychological traits make them vulnerable. Engineers don't understand user psychology too well. Fraudsters, however, do! Therefore engineers should learn from fraudsters, in order to build systems that are actually secure.

The Real Hustle[12], a BBC3 TV show by Paul Wilson and Alex Conran that aired for 11 series between 2006 and 2012, documented hundreds of actual scams and frauds[13], recreating them for hidden cameras. Were all these scams completely original or did they reuse a few basic ideas? Suspecting the latter, we set out to identify some kind of a "basis" for the "vector space of frauds". We identified seven principles that explain fundamental "system vulnerabilities" of the human psyche that fraudsters have been exploiting long before computers were invented. An understanding of these principles is necessary to build secure systems. Not just computer systems: any systems that involve people.

Our contributions: we documented existing scams; we extracted underlying principles; and we applied them to strengthen systems security.

Here is one example of the many scams we documented and analysed, the "Ring reward rip-off" (from Series 1, Episode 4[14]). The gorgeous Jess

[12] https://www.youtube.com/c/TheRealHustleOfficial/
[13] https://en.wikipedia.org/wiki/List_of_The_Real_Hustle_episodes
[14] https://youtu.be/Y8XSujAIUk8

buys a cheap ring from a market stall for 5 £. She then goes to a pub and seductively befriends the barman (the mark). She makes it obvious she's very rich; showing off to her friend (a shill), she makes sure the mark overhears that she just received this amazing 3,500 £ diamond ring for her birthday. She then leaves. Paul and Alex arrive at the pub, posing as two blokes having a pint. Jess then phones the pub, very worried, calling her friend the barman by name, saying she lost that very precious ring. Could he check if it's there somewhere? The mark checks and, luckily, a customer (Paul) found the ring. However, instead of handing it over, Paul demands a reward. The barman gets back to the phone and Jess, very relieved to hear the ring is there, says, without prompting, that she'll give 200 £ to the person who found it. But the barman goes back to Paul and says the reward is only 20 £. That's when the hustlers know they've got him; he's trying to make some profit for himself. Paul haggles a bit and eventually returns the ring to the barman for 50 £. The mark is all too happy to advance the money to Paul, expecting to get much more from Jess. But Jess, of course, never calls him back. A convicted criminal proudly says he once made a 2,000 £ profit with this particular hustle.

This scam relies primarily on what we call the Dishonesty principle. The mark (the barman) is lured into paying Paul because he hopes he will be cheating him out of a much larger reward that Jess promised. However the fact that he intended to cheat will make him unlikely to complain to the police once the reward never materialises, because he would have to confess to his own fraudulent intentions. This same principle is at the core of the very widespread computer-based fraud known as "419" or "Nigerian scam", in which criminals pose as family members of a wealthy prince and send the victims a litany of emails asking for assistance in transferring millions of dollars abroad, to the victims' country, in exchange for a generous proportion of it. But of course the victims are repeatedly asked to send some advance payments to "grease some wheels", while the actual transfer is postponed indefinitely. Once the victims realise they have been conned, they are reluctant to seek help from the police because they would have to confess to their complicity in money laundering and evasion of currency control regulations. Some victims have gone bankrupt and some have even committed suicide, seeing no way out of this dilemma.

Other principles at work in the Ring Reward Rip-Off include Need and Greed, whereby some urges, cravings and desires (including both the prospect of a monetary reward and the seductive attractiveness of Jess) become powerful ways that the hustlers can use to manipulate the victim; and the Time principle, whereby the fact that an opportunity will soon disappear forces the victim to decide quickly using fallible heuristics

rather than proper cool-headed reasoning.

We observed and documented hundreds of frauds, but almost all of them can be reduced to a handful of general principles that explain what victims fall for. The following table, from our 2009 tech report[15], illustrates which combination of our principles is exploited by each of the scams under analysis. Nowadays you may watch most of the TV episodes referred to below on the show's YouTube channel at https://www.youtube.com/@TheRealHustleOfficial.

	Episode	Scam	3.1 Distraction	3.2 Social Compliance	3.3 Herd	3.4 Dishonesty	3.5 Deception	3.6 Need and Greed	3.7 Time
2.1	S1-E1	Monte	●		●	○	○	○	○
2.2	S1-E2	Lottery scam			○	●	○	●	●
2.3	S1-E4	Ring reward rip-off				●	○	●	●
2.4	S2-E1	Van dragging	○	○			●		○
2.5	S2-E1	Home alone	○	●			○	○	
2.5.1	S1-E1	(Jewellery shop scam)	○	●			○	○	
2.6	S2-E3	Valet steal	○	●			●		○
2.7	S4-E3	Gadget scam				●	○	●	○
2.8	S4-E4	Counterfeit pen con		●			○		
2.9	S4-E5	Cash machine con	●		○		●		
2.10	S4-E5	Recruitment scam	●	●	○		●	○	
2.11	S4-E7	Shop phone call scam	○	●			○		○
2.12	S4-E9	Exchange rate rip-off	○		●		○	●	●

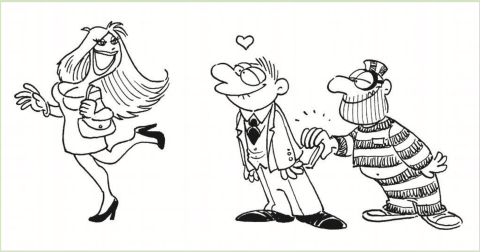
Our exact set of seven principles is not as important as the idea that almost all scams can be reduced to a few principles. The relevance of this idea to the security engineer is because these principles are also responsible for vulnerabilities in computer systems. They have been exploited by fraudsters for centuries before computers were invented. They are rooted in human nature. They are not going to go away. Acknowledge them, deal with their existence and abandon any hope to overcome them by issuing instructions to users.

These are the principles we distilled, graced by cute cartoons by my friend Silvia Ziche.

[15] http://www.cl.cam.ac.uk/techreports/UCAM-CL-TR-754.pdf

Dishonesty

Your larceny is what hooks you. Thereafter, anything illegal you do will be used against you by the fraudster.



Distraction

While you are distracted by what retains your interest, hustlers can do anything to you and you won't notice.

Herd

Even suspicious marks will let their guard down when others next to them appear to share the same risks. Safety in numbers? Not if they're all against you.

Kindness

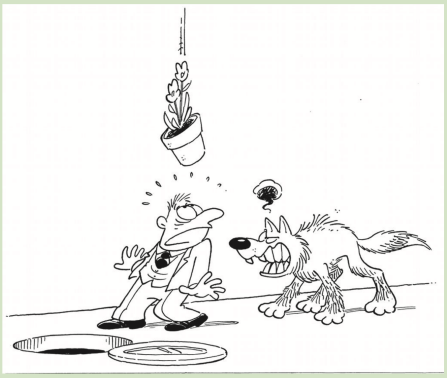
People are fundamentally nice and willing to help. Hustlers shamelessly take advantage of it.

Need and Greed

Need and greed make you vulnerable. Once hustlers know what you want, they can easily manipulate you.

Social Compliance

Society trains people not to question authority. Hustlers exploit this "suspension of suspiciousness" to manipulate you.



Time

When you are under time pressure to make an important choice, you use a different decision strategy. Hustlers steer you towards a strategy involving less reasoning.

After distilling our initial set of seven principles by observing fraudsters, we discovered striking similarities with the principles that Robert Cialdini[16] identified by observing salesmen. That's appropriate: after all, the techniques of fraudsters and pushy salesmen are very similar; perhaps the main difference is that, at least sometimes, what the salesmen do is legal... Another relevant work was by Stephen Lea et al, who

[16]Robert B. Cialdini. *Influence: science and practice*, Pearson, 1985; 5th edition 2009.

studied mass-marketed scams in the technical report[17] they prepared for the Office of Fair Trading.

Principles to which victims respond, as identified by three sets of researchers.

Principle	Cialdini (1985–2009)	Lea et al. (2009)	Stajano-Wilson (2009)
Distraction		~	•
Social Compliance (a.k.a. "Authority")	•	○	○
Herd (a.k.a. "Social Proof")	•		○
Dishonesty			•
Kindness	~		•
Need and Greed (a.k.a. "Visceral Triggers")	~	•	○
Scarcity (related to our "Time")	•	○	~
Commitment and Consistency	•	○	
Reciprocation	•		~

• First identified this principle
○ Also lists this principle
~ Lists a related principle

Our conclusion is that it is arrogantly idiotic for security engineers to whinge that "users are gullible". Certain behavioural patterns are simply human nature. Smart security engineers must acknowledge their inevitability and design the system to prevent their exploitation.

[17] Stephen E. G. Lea, Peter Fischer, Kath M. Evans (2009). "The psychology of scams: Provoking and committing errors of judgement". University of Exeter. Report. https://hdl.handle.net/10871/20958

Chapter 11

Malware

11.1 An incomplete taxonomy

Malware (malicious software) is a generic catch-all term for evil programs, that is to say any software that will damage computer systems or networks. The following definition of malware is from Cisco:

> Any intrusive software developed by cybercriminals (often called "hackers") to steal data and damage or destroy computers and computer systems. Examples of common malware include viruses, worms, Trojan viruses, spyware, adware, and ransomware.

The above taxonomy of categories of malware is not exhaustive and not everyone agrees on the precise definitions but it's still good to have at least a general idea of what these various nouns mean. Note the inconsistency in the above paragraph: sometimes the terms identify and classify the malware on the basis of the **attack vector**, or "how the malware gets into your system", as in the case of virus, worm and Trojan; whereas in other cases they refer to the **payload**, or "what the malware does to your system", as in the case of spyware, adware and ransomware. So you could reasonably have any combination of, at least, {virus, worm, Trojan} × {spyware, adware, ransomware}, for example a Trojan delivering some ransomware.

A **virus** is a piece of malware that attaches itself to executables and replicates itself, infecting other executables. When you run one of the programs on your system, if it has been infected it will first run the virus and then your original program. The virus, when executed, may behave in one of several ways. It may look for other executables and infect them. It may patch itself into the operating system and intercept system calls. It may do nothing, remaining stealthy until a certain trigger condition is

met; or it may activate its payload. The virus might be quite elaborate but it is just a vector, a method of propagation. What damage it causes to your system, for example deleting all your files, or installing a keylogger, or sending your banking credentials to a cybercriminal, depends on the payload.

There is a delicate balance, for the virus, between causing damage and propagating. A virus whose payload stays dormant for months[1] may spread rather more widely than one that alerts its victim to its presence, triggering immediate attempts at removal.

Self-replicating computer programs were first described by computer pioneer John von Neumann (1903–1957) back in 1949. The first viruses started circulating in the wild in the 1980s on the personal computers of the time, the Apple II and the IBM PC, as well as home computers such as Commodore Amiga and Atari ST. Hard disks were still expensive and comparatively rare among hobbyists, therefore those first viruses typically infected floppy disks, which were the most common storage medium. Floppies were frequently passed around among users and this was how the virus spread from machine to machine in the days before global networking. The fact that floppies were not permanently inserted into the machine made it really hard to eradicate an infection: it was always highly likely that a few infected disks would still lie around somewhere, ready to re-infect the system even after the most thorough of clean-ups.

The most fertile ground for viruses throughout the 1980s and 1990s were the MS-DOS and later MS Windows PCs, both because those were the most popular platforms and because those single-user operating systems running on processors without protected mode allowed any running program to do anything it wanted to the host machine and its peripherals. The first documented IBM PC virus, the so-called Brain, from Pakistan, appeared in 1986. A later development was the "Concept" macro virus of 1995: the introduction of Visual Basic macros into Microsoft Word and Excel had effectively transformed textual documents and spreadsheets into executables. Office employees sharing such files by email were unaware that they might also be spreading viruses.

The first antivirus programs relied on detecting "signatures", i.e. byte sequences that uniquely identified a particular virus. Virus authors responded by encrypting the code of the virus with a variable key, hiding the opcodes of the virus in a different way at each new infection. Antivirus authors, in turn, would detect the signature of the decryption engine, which had to remain in plaintext to do its job. The next counter-

[1] Such as the 1991 Michelangelo DOS virus, whose hard-disk-destroying payload would only activate on the 6th of March each year, the birthday of the eponymous Italian artist.

attack by virus writers consisted of polymorphic viruses, where the decryption engine itself would mutate at every infection. Antivirus authors then gradually switched to heuristic detection methods, looking not for ways of recognising previously observed instances of a virus but rather for behavioural patterns typical of viruses but uncommon in regular programs, such as "this program modifies itself". This arms race brings to mind the delightful "music to break phonographs by" in Douglas Hofstadter's masterpiece *Gödel, Escher, Bach* (1979), which predated these virus wars and yet insightfully addressed many of their most intellectually interesting aspects.

A **worm** is also a self-replicating piece of malware but, unlike a virus, it does not infect another executable and thus does not need a host program. A worm simply sends copies of itself through the network, infecting other connected machines. To run on remote hosts it may exploit common vulnerabilities. The first known worm was Creeper, written in 1971 to experiment with von Neumann's ideas on self-replicating programs. It spread through the ARPANET. It was followed by Reaper, designed to eradicate it. This chase in cyberspace was the inspiration for the 1984 Core Wars programming game, made popular by A. K. Dewdney in *Scientific American*. The first worm to spread extensively in the wild was the Morris Worm of 1988. It did not contain a destructive payload but it replicated so aggressively that it brought the infected machines to their knees just by overloading them. It caused a major outage on the pre-consumer Internet. To infect machines it relied on a variety of attack techniques we studied in this course, including exploiting a buffer overflow in `fingerd` and brute-forcing easy passwords. A worm that received substantial media coverage was Stuxnet, which appeared in 2010. It was a cyberweapon aimed at disrupting the Iranian nuclear programme by physically damaging Iranian uranium-enriching centrifuges, which it did on a large scale—it infected over 200,000 computers and destroyed 20% of Iran's nuclear centrifuges. Attributed by antivirus researchers to US and Israeli military, it exploited several zero-day[2] vulnerabilities in Microsoft Windows, but it initially bridged the air-gap to the Iranian nuclear facilities through infected USB sticks.

A **Trojan horse** (or often just "Trojan") is a piece of malware disguised as something else that the victim can be easily enticed to open (e.g. an attachment from a known contact, a letter promising fabulous benefits, a letter demanding money from the tax office, free pornographic material and so forth). The Trojan thus always propagates through social engineering. The payload of the Trojan, like that of the virus or

[2]Vulnerabilities never previously disclosed, for which security patches do not exist yet.

the worm, could be anything. The locution "Trojan virus" used in the Cisco quote above is an oxymoron because a virus would propagate by self-replication and infection of executables, whereas a Trojan horse is by definition propagated by tricking the victim into launching it[3]. By extension, dropping a malware-laden USB stick in the car park of one's victim, as Stuxnet apparently did to gain entry, can be considered a physical-world application of the Trojan horse technique.

Spyware is malware designed to sit silently on the victim's machine and allow the attacker to monitor what goes on, including potentially logging keystrokes and capturing passwords, turning on the camera and microphone, exfiltrating files and so forth. Spyware, like adware and ransomware, indicates a kind of payload, not a propagation vector: a piece of spyware could be dropped on your computer or phone by a variety of means, including as the payload of a virus, worm or Trojan. Spyware is generally designed to be stealthy so as to allow continued monitoring of the victim for an extended period of time. The sophisticated Pegasus spyware we mentioned in footnote 3 on page 18, first discovered in 2016, is a cyberweapon that attacks iOS and Android smartphones and that has been used by various oppressive regimes to spy on political opponents, dissidents, human rights activists and journalists.

Adware is malware that displays advertisements on your screen, usually after installing itself through subterfuge and without your consent. Some adware may include functionality bordering on spyware (user profiling for advertising purposes).

Ransomware is malware that encrypts your files, making them inaccessible to you, but promises to decrypt them back if you pay a ransom to the attacker in some untraceable way, usually involving cryptocurrency. The first known instance of malware that encrypted files and demanded a ransom was the AIDS Trojan of 1989[4] but it is only in the past decade that ransomware has become prominent (no doubt aided by the emergence and rise of bitcoin). The theoretical foundations of strong ransomware were laid in a 1996 article (well before bitcoin's appearance in 2009) by Adam Young and Moti Yung[5], who dubbed the

[3]It is true that a Trojan could, as its payload, drop a virus on the victim's machine, for example in order to spread further through the victim's organisation, but if we are just explaining the terms let's not get entangled into combinations—and in any case that's probably not what the author of the Cisco quote actually meant.

[4]This malware actually only encrypted the file names, not the files themselves, and with a symmetric key that a knowledgeable person could extract from the malware itself.

[5]Adam Young and Moti Yung, "Cryptovirology: extortion-based security threats and countermeasures". *Proceedings 1996 IEEE Symposium on Security and Privacy*, pp. 129–140. DOI: 10.1109/SECPRI.1996.502676. https://www.ieee-security.

attack *cryptoviral extortion*: once it runs on the victim's machine, the malware encrypts and overwrites the local files using a randomly generated symmetric key; then it encrypts that key under the public key of the crook and deletes the plaintext version of both the files and the symmetric key. It then prompts the victim to send the encrypted key to the crook, together with the ransom payment, in order to receive a usable decryption key by return. Under this scheme it is not possible for the victim to recover the decryption key by dissecting the malware. This type of cybercrime is on the rise. You might have heard of CryptoLocker (2013); of WannaCry (2017), which infected hundreds of thousands of computers in 150 countries and, in the UK, affected a number of NHS hospitals; and Petya, which in 2017 was used in high profile cyberattacks on Ukraine. The best safeguard against ransomware is to keep up-to-date offline backups, but this is a preventive recommendation that is of little use to someone who has just fallen victim to the attack.

Viruses and worms started out as intellectual curiosity that degenerated into mischief. By and large, during the 1980s and 90s, most malware was still only pointless vandalism (the virus that reformatted your hard disk) rather than a lucrative criminal activity. The start of the monetisation at scale of computer crime might arguably be traced back not to malware but to **phishing**, in the early 2000s, which we briefly discussed in Section 8.2 on page 109. Nowadays, "thanks" to a convergence of factors including the ubiquity of the internet and always-on networking, the emergence of e-commerce and online banking, and to some extent the anonymous payments made possible by decentralised cryptocurrencies, malware has graduated from mere vandalism to theft (banking keyloggers), extortion (ransomware), industrial or political espionage (spyware) and even cyber-weapons (attacks on industrial control systems and other critical infrastructure, of which the previously-mentioned Stuxnet was the flag-bearer).

11.2 Self-reproducing programs

From the viewpoint of computer science rather than that of criminology, perhaps the most intellectually interesting aspect of all the above is self-replication of programs. I keep telling you that to beat the bad guys we must be at least as smart as them, so let's try to be as smart as those who created the first viruses.

The virus must contain code that replicates itself. Can you write a program, in your favourite programming language, that prints out a

`org/TC/SP2020/tot-papers/young-1996.pdf`

copy of its source code on `stdout` when executed? That's a challenging puzzle. Douglas Hofstadter (1945–), of *Gödel, Escher, Bach* fame, called such self-reproducing programs "quines", in honour of logician Willard Van Orman Quine (1908–2000). It is instructive to invest some time in confronting this challenge without looking up a solution, and it is extremely rewarding to find one.

Your first few attempts will show you what doesn't work. Using Python 3, for example:

```
print('blah')
```

will output a disappointing

```
blah
```

and the equally unsuccessful

```
print('print("blah")')
```

will output

```
print("blah")
```

Obviously there will always be an extra layer of `print` in the source that this naïve strategy can never capture. A better strategy might be to build up a string variable containing the source code and printing that. Obviously the string cannot contain *all* of the source code...

```
source='source="stuff"; print(source)'; print(source)
```

...but if we could somehow substitute `stuff` with the `source` string itself then perhaps we might be getting somewhere. After all, Python offers you many option for replacing substrings, from `.replace()` to regular expressions to f-strings to percent-expansion. Can you make any progress from this hint[6]?

This ancient web page of mine https://www.cl.cam.ac.uk/~fms27/selfgen/, dating back to 1998 and therefore older than the undergraduate students currently taking this course, has a few more challenges along these lines (with obfuscated solutions to invite you to solve them on your own instead of peeking), including pairs of programs in different languages that, when run, print each other's source code. Have a go yourself!

[6]Resist the temptation to ask an LLM to do it for you—you'll receive a solution but you'll spoil forever your only opportunity to enjoy that priceless lightbulb moment.

Of course, solutions that *copy* their source code instead of generating a string from first principles must be considered cheating, as there is very little intellectual challenge in that. The Samy worm we discussed in section 9.1.2 on page 123 managed to access its own source through `document.body.innerHTML`, although Kamkar reports that Myspace attempted to preempt such exploits by stripping out the string "`innerHTML`" anywhere it found it. As you may recall if you read Kamkar's own write-up, the author of the Samy worm defeated that countermeasure by evaluating the concatenation of two substrings, "`inne`" and "`rHTML`". But what if Myspace had been successful in preventing the worm from accessing its own source? Would you be able to redo Task 6 of the SEED lab on XSS without being able to use either the Link or the DOM approach, but rather by self-generating the source? This would be a much harder challenge than any of the other SEED labs I have asked you to do, and I estimate that fewer than 5% of my Cambridge students could crack it. But I won't ask you to do it. I'm content with you coming up with a basic quine in your favourite programming language, provided you don't look it up anywhere else first. I guarantee that, once you succeed, you will be damn proud of yourself—and rightly so.

Chapter 12

Physical security

There is a natural affinity between the topics of cybersecurity and physical security. As I said in the introduction to this course, nobody can design a secure lock without being skilled at lockpicking—and this has an obvious direct parallel in the computer security world. As we say in Japanese swordsmanship, attack and defense are one.

Quite a few people with a serious interest in computer security also enjoy picking locks as a hobby. It's the same hacker[1] mindset. In any case, security is holistic: as we said on several occasions, we need to consider the whole system, and we need to use the broadest possible meaning for the word "system". In particular, physical security of the hardware cannot be underestimated. An attacker with physical control over your computer is generally in a position to do almost anything to it, including taking out the hard disk and imaging it, installing an invisible hardware keylogger that will henceforth exfiltrate all your keystrokes wirelessly, installing a hardware Trojan or just simply stealing the machine altogether, not to mention denial of service attacks involving sledgehammers[2]. It does therefore make sense to evaluate the physical security of the premises as part of a general security assessment and, towards that, some competence is required in, among other things, being able to discriminate between the locks that can be easily bypassed and those that put up a bit more of a fight.

People unfamiliar with lockpicking are often surprised when they discover how easy it often is to open what they thought was a good lock.

[1] Consult the hacker's dictionary (a.k.a. the Jargon File) to appreciate the difference between the in-group use of the term "hacker" and the mainstream press use that instead equates it with "computer criminal".

[2] And we're not even talking about what state-sponsored assassins might do, such as smearing polonium-210 (Litvinenko, 2006) or Novichok (Skripal, 2018) on the keyboard—but if you are in the crosshairs of such attackers you should have more serious concerns than merely computer security.

Very few locks cannot be opened by a skilled lockpicker equipped with good tools; but even a mediocre lockpicker will be able to get through rather more locks than you might think. Many products on the market are as carelessly produced, security-wise, as their software counterparts. And some of the most effective lock bypass techniques do not require great skill.

12.1 Attacks on pin-tumbler locks

The most common lock design is the pin-tumbler lock. A small cylinder called the **plug** must rotate for the lock to open, but is prevented from rotating by a row of pin stacks. Each pin stack contains two pins and a spring that pushes both pins towards the keyway. Of the two pins in each stack, the one in contact with the key (inside the plug) is called the key pin and the other, which is pushed into the plug by the spring, is known as the driver pin[3]. The plug may only rotate if, for each pin stack, the place where the two pins touch is perfectly aligned with the surface of the plug (the shear line). Otherwise, the pin that is intersecting the surface of the plug will prevent rotation. In normal operation, when the correct key is inserted, it lifts each key pin by the correct amount[4] to make it reach the shear line.

In theory, even though individual key pins can be manipulated by inserting a suitable tool (a **lockpick**) in the keyway, it should not be possible to check whether a certain pin height is correct for a given pin stack without having discovered the correct height for all the other pin stacks. If this theory were correct, the lockpicker would need to explore all combinations of pin heights, the number of which grows exponentially with the number of pin stacks. If s is the number of pin stacks and c is the number of different cut heights that a key may have in each position, the number of different keys in the system is c^s. The theory is that, in attempting to brute-force the lock, each of these combinations would have to be tried independently (either by cutting a new key or by somehow lifting each of the pins to the designated height simultaneously) and thus that the exponential expression c^s would quantify the effort of picking the lock.

[3]In the US, where the pin stacks are typically arranged *above* the key (key entering keyway with teeth upwards), the key pin is also known as the bottom pin and the driver pin is known as the top pin. But this is upside down with respect to European practice (key entering the keyway with teeth down) and thus, following Deviant Ollam, we prefer to use the unambiguous terminology of "key pin" and "driver pin" instead of "top and bottom pin"

[4]Potentially different for each pin.

Unfortunately for the defenders, though, the practice is rather different from the theory. Due to mechanical tolerances in the construction of the lock, if torque is applied to the plug without using the correct key, and the plug is therefore prevented from turning by the driver pins that are partly in the housing and partly in the plug, it is not the case that all s of the pins will be "binding" at once: instead, one of them will bind before the others, and the others will be comparatively loose. This effect is more pronounced in cheaper locks because of the poorer mechanical manufacturing tolerances. Given that, **single-pin picking** consists essentially of applying torque with a **tension tool** and attempting to push each pin stack gently against its spring ("lifting" the key pin) with a lockpick, until one stack is identified as resisting more than the others: that will be the one that is binding. If the pin stack in which the driver pin is binding is lifted, while simultaneously continuing to apply torque to the plug with a tensioning tool, on reaching the shear line the driver pin will exit the plug and at that point there will be a click, as the mechanical constraint that was stopping the plug from rotating is removed and the plug rotates by a minute angle until it stops against the next driver pin that is binding. Then the previous driver pin is considered "set": it is now completely outside the plug, and is prevented from being pushed back into the plug by the fact that that microscopic rotation has now partially obstructed its return path. The lockpicker then repeats the process with the remaining pin stacks that are not yet set, to identify the next driver pin that is binding. This goes on until the last pin stack is removed, at which point the plug will turn freely under the applied torque and the lock will open.

In terms of the complexity of the attack, the lockpicker must first probe all s pin stacks to find the one that is binding; then lift that one until it sets, potentially trying all the cut heights c; then probe the $s-1$ remaining pin stacks and lifting the binding one; then the remaining $s-2$ and so forth, in a process dominated by the number of probes which is the s^{th} triangular number, $\frac{s(s+1)}{2}$. In other words, the complexity of lockpicking using this single-pin-picking technique is only *quadratic* in s rather than exponential in s. And, when a skilled lockpicker does it, it takes less time to pick the lock than to read this description.

In *actual* practice, however, unless the pin tumbler lock is manufactured to very strict tolerances and/or contains special security pins, it will be susceptible to lockpicking attacks that require less skill and that are much faster. The **raking** attack consists of rapidly scrubbing all the pins back and forth with a jiggling motion of a wave-shaped lockpick, while applying torque to the plug with the tension tool. This operation essentially tries all the cases of single-pin-picking at high speed without

the lockpicker even being aware of which pin is binding at any given time. When it works, this technique opens the lock in seconds and it requires much less dexterity than single-pin picking.

If you become interested in locksport (the recreational activity of opening locks), raking will give you immediate satisfaction against your first few padlocks, and you'll learn this skill in no time. Single-pin picking will require a little more practice. Nowadays you can buy cheap pick sets from online marketplaces and also higher-quality lockpicks from specialised sellers. And you'll find a wealth of high quality lockpicking videos online: my top three favourite YouTube channels on the subject are those of LockPickingLawyer[5], Bosnian Bill[6] (now sadly retired, but all his videos are still up) and my friend Deviant Ollam[7], a physical penetration tester who has also written two outstanding books, *Practical lockpicking* and *Keys to the kingdom*. The most recent addition to my lockpicking bookshelf is the encyclopedic *Tobias on locks and insecurity engineering* by Mark Weber Tobias, whom I also interviewed on my channel[8]. But those of us who started lockpicking over three decades ago, before the web and YouTube and online stores existed, when the only available resource (via ftp) was the mythical *MIT guide to lockpicking*, used to make our own lockpicks by filing the steel bristles left behind by streetsweeper garbage trucks. Those were the days.

From the viewpoint of the security professional, though, you must understand that lockpicking is still, in a sense, just a game, with its own boundaries and rules, and that to assess the physical security of a system you must consider attackers that don't play by the rules of the game—any game. So how do they open a lock without even picking it?

Very few burglars actually engage in lockpicking, even at the relatively low skill level of raking. A technique that requires even less skill is **bumping**. A skeleton key is prepared, by cutting each key position to the greatest possible depth and by filing the collar slightly so that the key will penetrate the plug slightly more deeply than usual. Then the key is inserted and pushed in, so as to make contact with each of the key pins. Then the key is hit sharply with a rubber mallet or equivalent, while quickly applying torque a fraction of a second after hitting. What happens is that the impulse from the hit is transferred to the key pins (which cannot move because they are kept stuck in place by the pressure of the skeleton key that is jammed in) and, from those, to the driver pins, which jump out of the plug against the pressure of their springs.

[5] https://www.youtube.com/@lockpickinglawyer
[6] https://www.youtube.com/@bosnianbill
[7] https://www.youtube.com/@DeviantOllam
[8] https://youtu.be/S8Yf3Ok-FX8

If the torque is applied just at the right moment, when the driver pins have all flown out of the plug, the plug turns and the lock opens. It takes a little practice but this attack, when applicable, opens the lock almost instantaneously. The relatively few burglars that manipulate locks non-destructively tend to use bumping rather than lockpicking.

What most real-world burglars tend to do, however, is a rather more literal brute-force attack: **lock snapping**. They grab the cylinder with heavy-duty pliers and wiggle it sideways back and forth with great force until they snap it in two in the middle, at its weakest point where there is the least amount of metal. They then pull out the broken half of the cylinder and retract the latch manually. This is clearly a destructive attack, obviously not meant for covert entry, but it is low-skill and it lets them in in less than half a minute.

12.2 Countermeasures

Some locks will include "security pins" that, instead of being cylindrical, are spool-shaped, mushroom-shaped, serrated or otherwise weird. These interfere with the basic pin lifting process and are designed to get stuck at the shear line while the lockpicker applies torque. They will generally defeat raking and will make single-pin picking rather more difficult. A skilled lockpicker will still get past them, but they will stop the amateurs.

Springs of different strength in the various pin stacks are one countermeasure against bumping.

Regarding lock snapping, besides fitting a suitable escutcheon to the mouth of the lock to make it harder for the burglar to grab the cylinder with pliers, modern security cylinders have a sacrificial section at the front that will break off during a snapping attack but will leave the core of the lock still intact inside the door frame.

12.3 Privilege escalation in master lock systems

This fascinating piece of work[9] by security researcher and cryptographer Matt Blaze caused quite a stir in the locksmith community when it came out in 2002.

A master key system is one in which there are several locks with individual distinct keys (for the doors of the offices of the employees of

[9] Matt Blaze, "Rights amplification in master-keyed mechanical locks". *IEEE Security & Privacy*, vol. 1, no. 2, pp. 24-32, March-April 2003. DOI: 10.1109/MSECP.2003.1193208. https://www.mattblaze.org/papers/mk.pdf.

an organisation, say, where each employee can only open their own office but not those of the colleagues) and one master key for the building manager that opens all locks[10].

In pin tumbler locks, master keying is implemented by putting more than two pins in at least some of the pin stacks. If the possible cut depths are represented by integers between 0 (shallowest) and 9 (deepest) then a key in an s-pin lock is represented by an s-digit number, say 46217 for $s = 5$. This would be the employee key, normally indicated as a **differ key**. In this example, the second pin stack has a key pin corresponding to the cut depth of 6. If we replace it with a combination of a key pin of height 4 and a master pin of height 2, then their combined height will still be 6, meaning that the 46217 key will still open the lock; but now the 4**4**217 key (our **master key**) will open it as well.

If we wanted the master key to open all other doors in the building, how should we pin those other doors? Theoretically speaking[11], we have almost complete freedom: for each new lock we pick a previously unused combination of depths, say 34738, to be our differ key for that lock; and then we add the necessary cuts in the pin stacks to ensure the master key of 44217 also works. So here the key pins would be, respectively, (3+1), openable by 3 and 4; (4), openable just by 4; (2+5), openable by 2 and 7; (1+2), openable by 1 and 3; (7+1), openable by 7 and 8. This lock will thus be openable by both 34738 and 44217. But note the unintended and somewhat surprising side effect that it will also be openable by 14 more keys[12]! Thus, in a sense, the introduction of master keying has *reduced* the security of this lock, because there are fewer combinations to try before hitting one that works. Not many customers of master key systems realise this.

But that's not the worst of it, by far. Blaze's contribution was to point out a low complexity privilege escalation attack whereby a malicious insider, an "unprivileged user" entrusted with a differ key that only opened her office, could reconstruct the master key (the physical security equivalent of "getting root") with a search process of much greater efficiency than the exhaustive search of all combinations.

Blaze observed that, for each pin stack, if there is only one cut, then it must be the same for the differ key (owned and thus observable by

[10] I am simplifying here for the sake of clarity: more complex hierarchical arrangements of master keys are also possible.

[11] There are in fact some mechanical constraints to account for manufacturing tolerances but let's ignore that issue for now.

[12] You have two choices of valid cut depths for each of the pin stacks that have two cuts, for a total of $2^4 = 16$ different keys, including the master key and our designated differ key.

the unprivileged user) and the master key[13]; whereas, if there are two cuts, one is that of the differ key and the other must be the one of the master key. His crucial insight was that the unprivileged user can probe for this second cut on each of the pin stacks *separately*. How? Firstly, she prepares one key blank for each of the s pin stacks, and she cuts key blank i to the same depth as her differ key in every pin position except i, which she leaves uncut. Each of these $s = 5$ "attack keys" will be used to reveal the value of the cut of the master key in that position. For each attack key i, she files down pin position i gradually, exploring all possible cut depths from shallowest to deepest, one by one (they are discrete), at every step checking whether the key opens the lock. Of course she knows it will do so at the depth of her own differ key. If the lock also opens at another depth, that must be the depth of the cut of the master key in pin stack i. If no other cut depth opens the lock, then in that pin stack there is no master pin and the master key cut depth there is the same as that of her differ key. The complexity of this ingenious "physical privilege escalation" attack is linear (rather than exponential) in the number of pin stacks.

[13] As in the second pin from the left in our previous example of 34738 and 44217, where both keys had a 4.

Chapter 13

Conclusions

13.1 Security is risk management

If I had to describe the discipline of security in a nutshell, I would say that *security is risk management*. You identify the *assets* of the system that need protecting from undesirable events. You estimate the *value* of those assets to you (which gives you an upper bound on how much you should invest to protect them), and also to your *adversaries* (which contributes to your estimate of how motivated they are to go after those assets). You then identify the possible *threats* to your assets, i.e. the bad things that could happen to them; the possible *attacks*, i.e. the ways in which adversaries might actualise the threats; the *vulnerabilities*, i.e. the weaknesses in your system that make the attacks possible, or at least easier to carry out. You then consider the possible ways you might mitigate the possible attacks: some authors carefully distinguish between *safeguards* (preventive things you can do before disaster strikes, such as installing a stronger front door to your house, or taking backups of your digital assets) and *countermeasures* (a-posteriori remedies you can take after the fact, such as attempting to track down the thief, or setting a bounty, or retaliating[1]). Then, and this is the risk-management core of security, you do your best to quantify all those values. Given how much the asset is worth to you; given how great your loss would be for each of the corresponding threats to it; given how likely each of the attacks is, on the basis of both its value to the perpetrator and the difficulty of carrying it out given the existing vulnerabilities; given the cost of each of the safeguards and countermeasures... you apply some basic probability to compute your expected loss in the various alternative scenarios and you compare numerically whether paying for the certain and unrecoverable cost of the safeguards is better than incurring the loss on the assets if the

[1] Not always a good idea.

attacks succeed. You get the picture. This evaluation is never as precise as a mathematical formula based on this description might suggest, given that most of your input values are in fact only polite guesses; but the risk management viewpoint, imperfect though it may be, is still vastly superior to the Manichean and naïve viewpoint of those who would like the security consultant to name a price for "making the system totally secure". The price would have to be infinite, because the system can never be secure against *all* possible threats; but it would be silly to pay that price, or indeed any price that exceeds the value of the asset being protected—you would be better off allowing for the loss of the asset and then acquiring it again[2].

When you carry out this kind of probability calculation, keep an eye on the big picture and on the meaning and plausibility of the results, rather than accepting the numbers acritically. And when you are asked to make security recommendations, be aware that those who ask you to do so, and who may be your hierarchical superiors, may hold incorrect assumptions (perhaps influenced by what we discussed in chapter 10 on human factors, including for example prospect theory and framing effects). You will need to challenge those assumptions and, where necessary, debunk them.

If you retain only one thing from this course, let it be the above, along with the viewpoint that you must protect an entire *system* (including its users), not just a computer or the files on it, and that you must therefore take a holistic view and understand the true protection goals before embarking in the nitty-gritty technical detail—on which you must be more proficient than the attackers if you want a chance to defeat them.

I hope you enjoyed this course, whether you eventually become a security specialist or simply a better computer scientist. I hope it will inspire a few of you to dig deeper into security. We need more smart people like you in our field. If you were among the wise few who invested their hard-earned cash into buying the excellent W. Du, *Computer Security* 3rd ed textbook, you will find plenty more interesting material in the chapters we did not covered in this short course, and I encourage the more enterprising among you to explore those other chapters on your own. If security becomes your favourite topic, talk to me or to one of my colleagues about doing your final year undergraduate project, or your Master, or even your PhD, on a security-related topic. I am always on the lookout for outstanding students to work with. I am a demanding supervisor but there is good case history of my supervisees achieving excellence—from undergraduates who earned Firsts and won top-of-the-

[2] Although this is not possible for all assets—loss of life and loss of confidentiality cannot be reversed.

Tripos prizes, to PhDs who went on to professorships, at Cambridge and elsewhere. Some also earned well-paid part-time gigs at Cambridge Cyber. If you've got what it takes, you might be next.

13.2 Going forward

I invested a lot of work into preparing this course, recording the videos and writing these notes. While I welcome your low-level reports of typos, bugs and corrections, I value even more your higher level feedback: were there any topics you believe I didn't explain clearly enough, whether in this text or in the video lectures? Was there a SEED lab that you found too difficult, or one you believe I should have included instead of another? Were there any topics you believe I should expand on, or add, or remove? What parts, if any, did you particularly enjoy and think I should keep as they are? I might or might not agree with you on every one of your comments but all constructive feedback will be read carefully, taken into account and greatly appreciated. Write to me at the address on the cover page.

Best wishes for the rest of your Tripos, and for your career.