Introduction

Computer Security

By David Wagner, Nicholas Weaver, Peyrin Kao, Fuzail Shakir, Andrew Law, and Nicholas Ngai

Additional contributions by Noura Alomar, Sheqi Zhang, and Shomil Jain

This is the textbook for CS 161: Computer Security at UC Berkeley. It provides a brief survey over common topics in computer security including memory safety, cryptography, web security, and network security.

PDF Version

These notes are available as a PDF.

This PDF is automatically generated each time there is a correction made to the website. This means that it is *completely possible* for a downloaded copy to be not up-to-date. If your desired mode of the textbook is in PDF form, we recommend you regularly redownload in cases there were recent corrections made.

Corrections

As of the Fall 2024 semester, this textbook is still being actively maintained and updated.

If you see any parts that needs to be corrected, please open a Github issue here.

Source and Changelog

The source for the textbook and a log of all changes is available on Github.

License

Security Principles

Security Principles

In this section, we will look at some general principles for secure system design. These ideas also allow us to examine existing systems to understand their security properties.

In other words, this section contains a list of "things to remember" when thinking about security.

We teach these security principles because they appear frequently in all aspects of the security field. You may hear about them in academic literature and in later parts of this class.

1. Security Principles

1. Security Principles

1.1. Know your threat model

A threat model is a model of who your attacker is and what resources they have. Attackers target systems for various reasons, be it money, politics, fun, etc. Some aren't looking for anything logical–some attackers just want to watch the world burn.

Take, for example your own personal security. Understanding your threat model has to do with understanding who and why might someone attack you; criminals, for example, could attack you for money, teenagers could attack you for laughs (or to win a dare), governments might spy on you to collect intelligence (but you probably are not important enough for that just yet), or intimate partners could spy on you.

Once you understand who your attacker is and what resources they might possess, there are some common assumptions that we take into account for attackers:

1. The attacker can interact with your systems without anyone noticing, meaning that you might not always be able to detect the attacker tampering with your system before they attack.
2. The attacker has some general information about your system, namely the operating system, any potential software vulnerabilities, etc.
3. The attacker is persistent and lucky; for example, if an attack is successful 1/1,000,000 times, the attacker will try 1,000,000 times.
4. The attacker has the resources required to undertake the attack (up to an extent). This will be touched on in "Securities is Economics", but depending on who your threat model is, assume that the attacker has the ability and resources to perform the attack.
5. The attacker can coordinate several complex attacks across various systems, meaning that the attacker does not have to mount only a single attack on one device, but rather can attack your entire network at the same time.
6. Every system is a potential target. For example, a casino was once hacked because a fish-tank thermometer was hacked within the network.

Finally, be extremely vigilant when dealing with old code as the assumptions that were originally made might no longer be valid and the threat model might have changed. When the Internet was first created, for example, it was mostly populated by academics who (mostly) trusted one another. As such, several networking protocols made the assumption that all other network participants could be trusted and were not malicious. Today however, the Internet is populated by billions of devices, some of whom are malicious. As such, many network protocols that were designed a long time ago are now suffering under the strain of attack.

1.2. Consider Human Factors

The key idea here is that security systems must be usable by ordinary people, and therefore must be designed to take into account the role that humans will play. As such, you must remember that programmers make mistakes and will use tools that allow them to make mistakes (like C and C++). Similarly, users like convenience; if a security system is unusable and not user-friendly, no matter how secure it is, it will go unused. Users will find a way to subvert security systems if it makes their lives easier.

No matter how secure your system is, it all comes down to people. Social engineering attacks, for example, exploit other people's trust and access for personal gain. The takeaway here is to consider the tools that are presented to users, and try to make them fool-proof and as user-friendly as possible.

For example, your computer pops up with a notification that tells you it needs to restart to "finish installing important updates"; if you are like a majority of the user population, you likely click "remind me later", pushing off the update. If the computer is attempting to fix a security patch, the longer the update gets pushed, the more time your computer is vulnerable to an attack. However, since the update likely inconveniences the user, they forego the extra security for convenience.

Another example: the NSA's cryptographic equipment stores its key material on a small physical token. This token is built in the shape of an ordinary door key. To activate an encryption device, you insert the key into a slot on the device and turn the key. This interface is intuitively understandable, even for 18-year-old soldiers out in the field with minimal training in cryptography.

1.3. Security is economics

No system is completely, 100% secure against all attacks; rather, systems only need to be protected against a certain level of attacks. Since more security usually costs more money to implement, the expected benefit of your defense should be proportional to the expected cost of the attack. Essentially, there is no point putting a $100 lock on a $1 item.

To understand this concept, we can think about physical safes, which come with a rating of their level of security. For instance, a consumer grade safe, a TL-15, might indicate that it will resist attacks for up to 15 minutes by anyone with common tools, and might cost around $3,000, while a TL-30, a safe that would resist attacks for up to 30 minutes with common tools might cost around $5000. Finally, a TXTL-60 (a super high-end safe), might resist attacks for up to 60 minutes with common tools, a cutting torch, and up to 4 oz of explosives, and would cost upwards of $50,000. The idea is that security usually comes at a cost. A more secure safe is going to cost you more than a less secure safe. With infinite money, you could use the best safe available to lock all your valuables, but since you don't have infinite money, you must determine how valuable the thing you want to protect is, and you must judge how much you are willing to pay to protect it. This illustrates that security is often a cost-benefit analysis where someone needs to make a decision regarding how much security is worth.

A corollary of this principle is you should focus your energy on securing the weakest links. Security is like a chain: a system is only as secure as the weakest link. Attackers follow the path of least resistance, and they will attack the system at its weakest point. There is no sense putting an expensive high-end deadbolt on a screen door; attackers aren't going to bother trying to pick the lock when they can just rip out the screen and step through.

A closely related principle is conservative design, which states that systems should be evaluated according to the worst security failure that is at all plausible, under assumptions favorable to the attacker. If there is any plausible circumstance under which the system can be rendered insecure, then it is prudent to consider seeking a more secure system. Clearly, however, we must balance this against "security is economics": that is, we must decide the degree to which our threat model indicates we indeed should spend resources addressing the given scenario.

1.4. Detect if you can't prevent

If prevention is stopping an attack from taking place, detection is simply learning that the attack has taken place, and response would be doing something about the attack. The idea is that if you cannot prevent the attack from happening, you should at least be able to know that the attack has happened. Once you know that the attack has happened, you should find a way to respond, since detection without response is pointless.

For example, the Federal Information Processing Standard (FIPS) are publicly announced standards developed for use in computer systems by various government contractors. Type III devices—the highest level of security in the standard, are intended to be tamper-resistant. However, Type III devices are very expensive. Type II

devices are only required to be tamper-evident, so that if someone tampers with them, this will be visible (e.g., a seal will be visibly broken). This means they can be built more cheaply and used in a broader array of applications.

When dealing with response, you should always assume that bad things will happen, and therefore prepare your systems for the worst case outcome. You should always plan security in a way that lets you get back to some form of a working state. For example, keeping offsite backups of computer systems is a great idea. Even if your system is completely destroyed, it should be no big deal since all your data is backed up in some other location.

1.5. Defense in depth

The key idea of defense in depth is that multiple types of defenses should be layered together so an attacker would have to breach all the defenses to successfully attack a system.

Take, for example, a castle defending its king. The castle has high walls. Behind those walls might be a moat, and then another layer of walls. Layering multiple simple defensive strategies together can make security stronger. However, defense in depth is not foolproof–no amount of walls will stop siege cannons from attacking the castle. Also, beware of diminishing returns–if you've already built 100 walls, the 101st wall may not add enough additional protection to justify the cost of building it (security is economics).

Another example of defense in depth is through a composition of detectors. Say you had two detectors, D_1 and D_2, which have false positive rates of FP_1 and FP_2 respectively, and false negative rates of FN_1 and FN_2, respectively. One way to use the two detectors would be to have them in parallel, meaning that either detector going off would trigger a response. This would increase the false positive rate and decrease the false negative rate. On the other hand, we could also have the detectors in series, meaning that both detectors have to alert in order to trigger a response. In this case, the false positive rate would decrease while the false negative rate would increase.

1.6. Least privilege

Consider a research building home to a team of scientists as well as other people hired to maintain the building (janitors, IT staff, kitchen staff, etc.) Some rooms with sensitive research data might be only accessible to trusted scientists. These rooms should not be accessible to the maintenance staff (e.g. janitors). For best security practices, any one party should only have as much privilege as it needs to play its intended role.

In technical terms, give a program the set of access privileges that it legitimately needs to do its job—but nothing more. Try to minimize how much privilege you give each program and system component.

Least privilege is an enormously powerful approach. It doesn't reduce the probability of failure, but it can reduce the expected cost of failures. The less privilege that a program has, the less harm it can do if it goes awry or becomes subverted.

For instance, the principle of least privilege can help reduce the damage caused by buffer overflow. (We'll discuss buffer overflows more in the next section.) If a program is compromised by a buffer overflow attack, then it will probably be completely taken over by an intruder, and the intruder will gain all the privileges the program had. Thus, the fewer privileges that a program has, the less harm is done if it should someday be penetrated by a buffer overflow attack.

How does Unix do, in terms of least privilege? Answer: Pretty lousy. Every program gets all the privileges of the user that invokes it. For instance, if I run a editor to edit a single file, the editor receives all the privileges of my user account, including the powers to read, modify, or delete all my files. That's much more than is needed; strictly speaking, the editor probably only needs access to the file being edited to get the job done.

How is Windows, in terms of least privilege? Answer: Just as lousy. Arguably worse, because many users run under an Administrator account, and many Windows programs require that you be Administrator to run them. In this case, every program receives total power over the whole computer. Folks on the Microsoft

security team have recognized the risks inherent in this, and have taken many steps to warn people away from running with Administrator privileges, so things have gotten better in this respect.

1.7. Separation of responsibility

Split up privilege, so no one person or program has complete power. Require more than one party to approve before access is granted.

In a nuclear missile silo, for example, two launch officers must agree before the missile can be launched.

Another example of this principle in action is in a movie theater, To watch a movie, you first pay the cashier and get a ticket stub. Then, when you enter the movie theater, a different employee tears your ticket in half and collects one half of it, putting it into a lockbox. Why bother giving you a ticket that 10 feet later is going to be collected from you? One answer is that this helps prevent insider fraud. Employees might be tempted to let their friends watch a movie without paying. The presence of two employees makes an attack harder, since both employees must work together to let someone watch a movie without paying.

In summary, if you need to perform a privileged action, require multiple parties to work together to exercise that privilege, since it is more likely for a single party to be malicious than for all of the parties to be malicious and collude with one another.

1.8. Ensure complete mediation

When enforcing access control policies, make sure that you check *every* access to *every* object. This kind of thinking is helpful to detect where vulnerabilities could be. As such, you have to ensure that all access is monitored and protected. One way to accomplish this is through a *reference monitor*, which is a single point through which all access must occur.

1.9. Shannon's Maxim

Shannon's Maxim states that the attacker knows the system that they are attacking.

"Security through obscurity" refers to systems that rely on the secrecy of their design, algorithms, or source code to be secure. The issue with this, however, is that it is extremely brittle and it is often difficult to keep the design of a system secret from a sufficiently motivated attacker. Historically, security through obscurity has a lousy track record: many systems that have relied upon the secrecy of their code or design for security have failed miserably.

In defense of security through obscurity, one might hear reasoning like: "this system is so obscure, only 100 people around the world understand anything about it, so what are the odds that an adversary will bother attacking it?" One problem with such reasoning is that such an approach is self-defeating. As the system becomes more popular, there will be more incentive to attack it, and then we cannot rely on its obscurity to keep attackers away.

This doesn't mean that open-source applications are necessarily more secure than closed-source applications. But it does mean that you shouldn't trust any system that *relies* on security through obscurity, and you should probably be skeptical about claims that keeping the source code secret makes the system significantly more secure.

As such, you should never rely on obscurity as part of your security. Always assume that the attacker knows every detail about the system that you are working with (including its algorithms, hardware, defenses, etc.)

A closely related principle is Kerckhoff's Principle, which states that cryptographic systems should remain secure even when the attacker knows all internal details of the system. (We'll discuss cryptographic systems more in the cryptography section.) The secret key should be the only thing that must be kept secret, and the system should be designed to make it easy to change keys that are leaked (or suspected to be leaked). If your secrets are leaked, it is usually a lot easier to change the key than to replace every instance of the running software.

1.10. Use fail-safe defaults

Choose default settings that "fail safe", balancing security with usability when a system goes down. When we get to firewalls, you will learn about default-deny polices, which start by denying all access, then allowing only those which have been explicitly permitted. Ensure that if the security mechanisms fail or crash, they will default to secure behavior, not to insecure behavior.

For example, firewalls must explicitly decide to forward a given packet or else the packet is lost (dropped). If a firewall suffers a failure, no packets will be forwarded. Thus, a firewall fails safe. This is good for security. It would be much more dangerous if it had fail-open behavior, since then all an attacker would need to do is wait for the firewall to crash (or induce a crash) and then the fort is wide open.

1.11. Design security in from the start

Trying to retrofit security to an existing application after it has already been spec'ed, designed, and implemented is usually a very difficult proposition. At that point, you're stuck with whatever architecture has been chosen, and you don't have the option of decomposing the system in a way that ensures least privilege, separation of privilege, complete mediation, defense in depth, and other good properties. Backwards compatibility is often particularly painful, because you can be stuck with supporting the worst insecurities of all previous versions of the software.

Finally, let's examine three principles that are widely accepted in the cryptographic community (although not often articulated) that can play a useful role in considering computer system security as well.

1.12. The Trusted Computing Base (TCB)

Now that you understand some of the important principles for building secure systems, we will try to see what you can do at design time to implement these principles and improve security. The question we want to answer is how can you choose an architecture that will help reduce the likelihood of flaws in your system, or increase the likelihood that you will be able to survive such flaws? We begin with a powerful concept, the notion of a trusted computing base, also known as the TCB.

In any system, the *trusted computing base* (TCB) is that portion of the system that must operate correctly in order for the security goals of the system to be assured. We have to rely on every component in the TCB to work correctly. However, anything that is outside the TCB isn't relied upon in any way; even if it misbehaves or operates maliciously, it cannot defeat the system's security goals. Generally, the TCB is made to be as small as possible since a smaller, simpler TCB is easier to write and audit.

Suppose the security goal is that only authorized users are allowed to log into my system using SSH. What is the TCB? Well, the TCB includes the SSH daemon, since it is the one that makes the authentication and authorization decisions; if it has a bug, or if it was programmed to behave maliciously, then it will be able to violate my security goal by allowing access to unauthorized users. The TCB also includes the operating system, since the operating system has the power to tamper with the operation of the SSH daemon (e.g., by modifying its address space). Likewise, the CPU is in the TCB, since we are relying upon the CPU to execute the SSH daemon's machine instructions correctly. Suppose a web browser application is installed on the same machine; is the web browser in the TCB? Hopefully not! If we've built the system in a way that is at all reasonable, the SSH daemon is supposed to be protected (by the operating system's memory protection) from interference by unprivileged applications, like a web browser.

TCB Design Principles: Several principles guide us when designing a TCB:

- *Unbypassable (or completeness):* There must be no way to breach system security by bypassing the TCB.
- *Tamper-resistant (or security):* The TCB should be protected from tampering by anyone else. For instance, other parts of the system outside the TCB should not be able to modify the TCB's code or state. The integrity of the TCB must be maintained.

- *Verifiable (or correctness):* It should be possible to verify the correctness of the TCB. This usually means that the TCB should be as simple as possible, as generally it is beyond the state of the art to verify the correctness of subsystems with any appreciable degree of complexity.

Keeping the TCB **simple and small** is excellent. The less code you have to write, the fewer chances you have to make a mistake or introduce some kind of implementation flaw. Industry standard error rates are 1–5 defects per thousand lines of code. Thus, a TCB containing 1,000 lines of code might have 1–5 defects, while a TCB containing 100,000 lines of code might have 100–500 defects. If we need to then try to make sure we find and eliminate any defects that an adversary can exploit, it's pretty clear which one to pick![1] The lesson is to shed code: design your system so that as much code as possible can be *moved outside* the TCB.

Benefits of TCBs: The notion of a TCB is a very powerful and pragmatic one as it allows a primitive yet effective form of modularity. It lets us separate the system into two parts: the part that is security-critical (the TCB), and everything else.

This separation is a big win for security. Security is hard. It is really hard to build systems that are secure and correct. The more pieces the system contains, the harder it is to assure its security. If we are able to identify a clear TCB, then we will know that only the parts in the TCB must be correct for the system to be secure. Thus, when thinking about security, we can focus our effort where it really matters. And, if the TCB is only a small fraction of the system, we have much better odds at ending up with a secure system: the less of the system we have to rely upon, the less likely that it will disappoint.

In summary, some good principles are:

- Know what is in the TCB. Design your system so that the TCB is clearly identifiable.
- Try to make the TCB unbypassable, tamper-resistant, and as verifiable as possible.
- Keep It Simple, Stupid (KISS). The simpler the TCB, the greater the chances you can get it right.
- Decompose for security. Choose a system decomposition/modularization based not just on functionality or performance grounds—choose an architecture that makes the TCB as simple and clear as possible.

1.13. TOCTTOU Vulnerabilities

A common failure of ensuring complete mediation involves race conditions. The time of check to time of use (TOCTTOU) vulnerability usually arises when enforcing access control policies such as when using a reference monitor. Consider the following code:

```
procedure withdraw(amount w) {
    // contact central server to get balance
    1. let b := balance
    2. if b < w, abort

    // contact central server to set the balance
    3. set balance := b - w
    4. give w dollars to the user
}
```

This code takes as input the amount you want to withdraw, w. It then looks up your bank balance in the database; if you do not have enough money in your account to withdraw the specified amount, then it aborts the transaction. If you do have enough money, it decrements your balance by the amount that you want to withdraw and then dispenses the cash to you.

Suppose that multiple calls to withdraw can take place concurrently (i.e. two separate ATMs). Also suppose that the attacker can somehow pause the execution of procedure on one ATM.

So suppose that your current account balance is $100 and you want to withdraw $100. At the first ATM, suppose you pause it *after* step 2. Then, you go over to the second ATM and proceed to withdraw $100 successfully (meaning that your account balance should now be $0). You then go back to the first ATM and

[1] Windows XP consisted of about 40 million lines of code—all of which were in the TCB. Yikes!

unpause the procedure; since the account balance check was completed before you withdrew the money from the second ATM, the first ATM still thinks you have $100 in your account, and it allows you to withdraw another $100! So despite your bank account having only $100 to begin with, you ended up with $200.

This is known as a *Time-Of-Check To Time-Of-Use* (TOCTTOU) vulnerability, because between the check and the use of whatever state was checked, the state somehow changed. In the above example, between the time that the balance was checked and the time that balance was set, the balance was somehow changed.

Memory Safety

Memory Safety

In this section, we will be looking at software security—problems associated with the software implementation. You may have a perfect design, a perfect specification, perfect algorithms, but still have implementation vulnerabilities. In fact, after configuration errors, implementation errors are probably the largest single class of security errors exploited in practice.

In particular, we will look at a particularly prevalent class of software flaws, those that concern *memory safety*. Memory safety refers to ensuring that attackers cannot read or write to memory locations other than those intended by the programmer.

Because many security-critical applications have been written in C, and because C is not a memory-safe language, we will focus on memory safety vulnerabilities and defenses in C.

2. x86 Assembly and Call Stack

2. x86 Assembly and Call Stack

This section reviews some relevant concepts from CS 61C and introduces x86 assembly, which is different from the RISC-V assembly taught in 61C.

2.1. Number representation

At the lowest level, computers store memory as individual bits, where each bit is either 0 or 1. There are several units of measurement that we use for collections of bits:

- 1 *nibble* = 4 bits
- 1 *byte* = 8 bits
- 1 *word* = 32 bits (on 32-bit architectures)

A "word" is the size of a pointer, which depends on your CPU architecture. Real-world 64-bit architectures often include stronger defenses against memory safety exploits, so for ease of instruction, this class uses 32-bit architectures unless otherwise stated.

For example, the string `1000100010001000` has 16 bits, or 4 nibbles, or 2 bytes.

Sometimes we use hexadecimal as a shorthand for writing out long strings of bits. In hexadecimal shorthand, a nibble can be written as a single hexadecimal digit. The chart below shows conversions between nibbles written in binary and hexadecimal.

Binary	Hexadecimal	Binary	Hexadecimal
0000	0	1000	8
0001	1	1001	9
0010	2	1010	A
0011	3	1011	B
0100	4	1100	C
0101	5	1101	D
0110	6	1110	E
0111	7	1111	F

To distinguish between binary and hexadecimal strings, we put `0b` before binary strings and `0x` before hexadecimal strings.

Sanity check: Convert the binary string `0b1100000101100001` into hexadecimal.[1]

2.2. Compiler, Assembler, Linker, Loader (CALL)

Recall from 61C that there are four main steps to running a C program.

[1] Answer: Using the table to look up each sequence of 4 bits, we get `0xC161`.

1. The *compiler* translates your C code into assembly instructions. 61C uses the RISC-V instruction set, but in 161, we use x86, which is more commonly seen in the real world.
2. The *assembler* translates the assembly instructions from the compiler into machine code (raw bits). You might remember using the RISC-V green sheet to translate assembly instructions into raw bits in 61C. This is what the assembler does.
3. The *linker* resolves dependencies on external libraries. After the linker is finished linking external libraries, it outputs a binary executable of the program that you can run (you can mostly ignore the linker for the purposes of 161).
4. When the user runs the executable, the *loader* sets up an address space in memory and runs the machine code instructions in the executable.

2.3. C memory layout

At runtime, the operating system gives the program an address space to store any state necessary for program execution. You can think of the address space as a large, contiguous chunk of memory. Each *byte* of memory has a unique address.

The size of the address space depends on your operating system and CPU architecture. In a 32-bit system, memory addresses are 32 bits long, which means the address space has 2^{32} bytes of memory. In a 64-bit system, memory addresses are 64 bits long. (Sanity check: how big is the address space in this system?[2]) In this class, unless otherwise stated we'll be using 32-bit systems.

We can draw the memory layout as one long array with 2^{32} elements, where each element is one byte. The leftmost element has address `0x00000000`, and the rightmost element has address `0xFFFFFFFF`.[3]

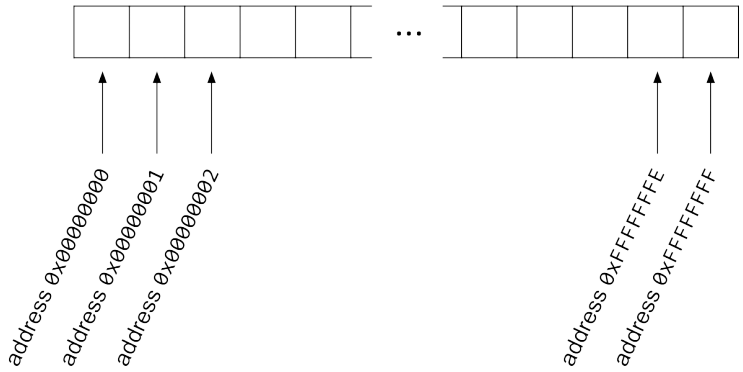

Figure 1: 1-dimensional address space

However, this is hard to read, so we usually draw memory as a grid of bytes. In the grid, the bottom-left element has address `0x00000000`, and the top-right element has address `0xFFFFFFFF`. Addresses increase as you move from left to right and from bottom to top.

Although we can draw memory as a grid with annotations and labels, remember that the program only sees a huge array of raw bytes. It is up to the programmer and the compiler to manipulate this chunk of raw bytes to create objects like variables, pointers, arrays, and structs.

When a program is being run, the address space is divided into four sections. From lowest address to highest address, they are:

- The *code* section contains the executable instructions of the program (i.e. the code itself). Recall that the assembler and linker output raw bytes that can be interpreted as machine code. These bytes are stored in the code section.

[2] Answer: 2^{64} bytes.

[3] In reality your program may not have all this memory, but the operating system gives the program the illusion that it has access to all this memory. Refer to the virtual memory unit in CS 61C or take CS 162 to learn more.

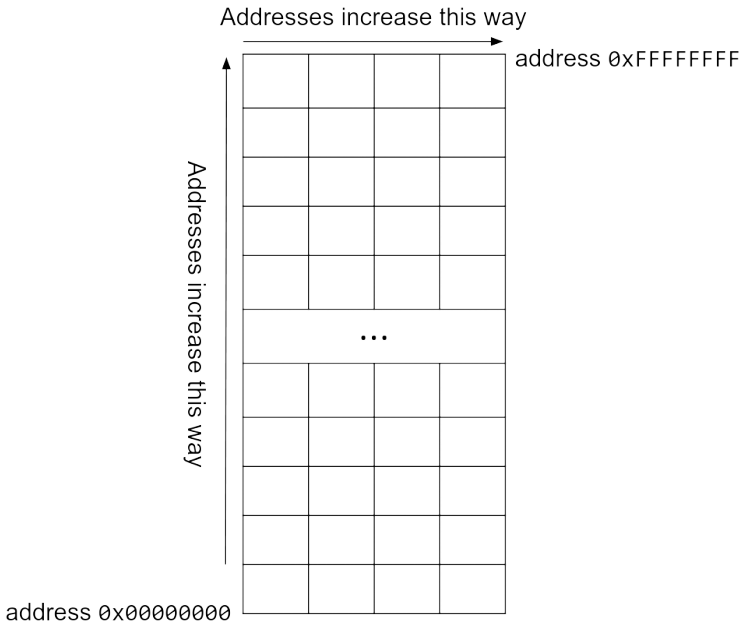

Figure 2: 2-dimensional address space

- The *static* section contains constants and static variables that never change during program execution, and are usually allocated when the program is started.
- The *heap* stores dynamically allocated data. When you call `malloc` in C, memory is allocated on the heap and given to you for use until you call `free`. The heap starts at lower addresses and "grows up" to higher addresses as more memory is allocated.
- The *stack* stores local variables and other information associated with function calls. The stack starts at higher addresses and "grows down" as more functions are called.

2.4. Little-endian words

x86 is a *little-endian* system. This means that when storing a word in memory, the least significant byte is stored at the lowest address, and the most significant byte is stored at the highest address. For example, here we are storing the word `0x44332211` in memory:

Note that the least significant byte `0x11` is stored at the lowest address, and the most significant byte `0x44` is stored at the highest address.

Because we work with words so often, sometimes we will write words on the memory diagram instead of individual bytes. Each word is 4 bytes, so each row of the diagram has exactly one word.

Using words on the diagram lets us abstract away little-endianness when working with memory diagrams. However, it's important to remember that the bytes are actually being stored in little-endian format.

2.5. Registers

In addition to the 2^{32} bytes of memory in the address space, there are also *registers*, which store memory directly on the CPU. Each register can store one word (4 bytes). Unlike memory, registers do not have addresses. Instead, we refer to registers using names. There are three special x86 registers that are relevant for these notes:

- *eip* is the *instruction pointer*, and it stores the address of the machine instruction currently being executed. In RISC-V, this register is called the PC (program counter).

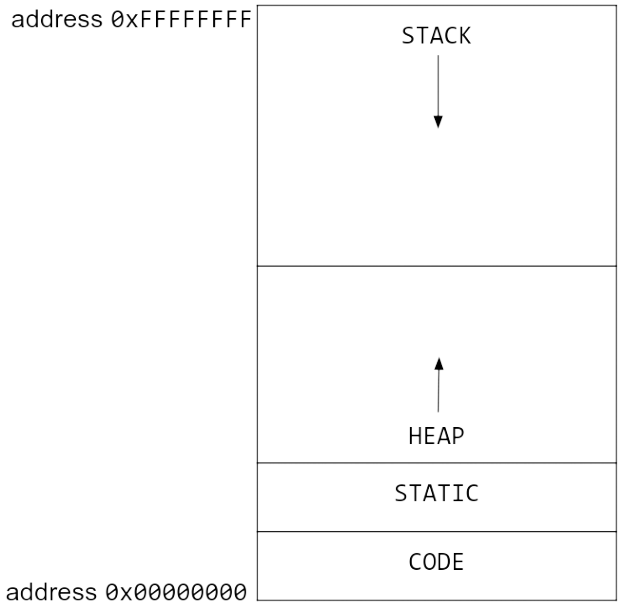

Figure 3: Memory sections

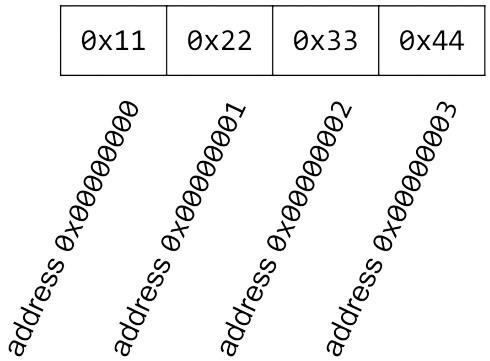

Figure 4: Little-endian word format

- *ebp* is the *base pointer*, and it stores the address of the top of the current stack frame. In RISC systems, this register is called the `FP` (frame pointer)[4].
- *esp* is the *stack pointer*, and it stores the address of the bottom of the current stack frame. In RISC-V, this register is called the `SP` (stack pointer).

Note that the top of the current stack frame is the highest address associated with the current stack frame, and the bottom of the stack frame is the lowest address associated with the current stack frame.

If you're curious, the e in the register abbreviations stands for "extended" and indicates that we are using a 32-bit system (extended from the original 16-bit systems).

Since the values in these three registers are usually addresses, sometimes we will say that a register *points* somewhere in memory. This means that the address stored in the register is the address of that location in memory. For example, if we say eip is pointing to `0xDEADBEEF`, this means that the eip register is storing the value `0xDEADBEEF`, which can be interpreted as an address to refer to a location in memory.

Sanity check: Which section of C memory (code, static, heap, stack) do each of these registers usually point to?[5]

2.6. Stack: Pushing and popping

Sometimes we want to remember a value by saving it on the stack. There are two steps to adding a value on the stack. First, we have to allocate additional space on the stack by decrementing the esp. Then, we store the value in the newly allocated space. The x86 `push` instruction does both of these steps to add a value to the stack.

Figure 5: Before and after of pushing an item onto the stack

We may also want to remove values from the stack. The x86 `pop` instruction increments esp to remove the next value on the stack. It also takes the value that was just popped and copies the value into a register.

Note that when we pop a value off the stack, the value is not wiped away from memory. However, we increment esp so that the popped value is now below esp. The esp register points to the bottom of the stack, so the popped value below esp is now in undefined memory.

(eax and ebx are general-purpose registers in x86. We use them here as an example of pushing and popping from the stack, but you don't need to know anything else about these registers.)

[4]RISC systems often omit this register because it is not necessary with the RISC stack design. For example, in RISC-V, `FP` is sometimes renamed `s0` and used as a general-purpose register

[5]Answer: eip points to the code section, where instructions are stored. ebp and esp point to the stack section.

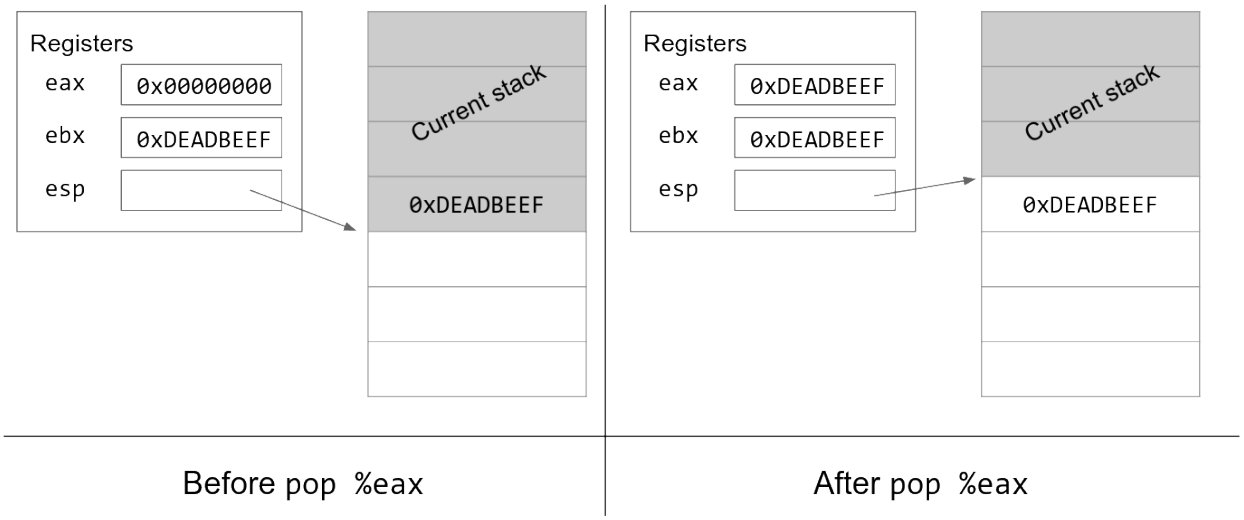

Figure 6: Before and after of popping an item off the stack

2.7. x86 calling convention

This class uses AT&T x86 syntax (since that is what GDB uses). This means that the destination register comes last; note that this is in contrast with RISC-V assembly, where the destination register comes first. Suppose our assembly instruction was `addl $0x8, %ebx`; here, the opcode is `addl`, the source is `$0x8`, and the destination is `%ebx`, so in pseudocode this can be read as `EBX = EBX + 0x8`.

References to registers are preceded with a percent sign, so if we wanted to reference `eax`, we would do so as `%eax`. Immediates are preceded with a dollar sign (i.e. $1, $0x4, etc.). Furthermore, memory references use parenthesis and can have immediate offsets; for example, `12(%esp)` dereferences memory 12 bytes above the address contained in ESP. If parentheses are used without an immediate offset, the offset can be thought of as an implicit 0.

Suppose our assembly instruction was `xorl 4(%esi), %eax`; here, the opcode is `xorl`, the source is `4(%esi)`, and the destination is `%eax`. As such, in pseudocode, this can be written as `EAX = EAX ^ *(ESI + 4)`. Since this is a memory reference, we are dereferencing the value 4 bytes above the address stored in ESI.

2.8. x86 function calls

When a function is called, the stack allocates extra space to store local variables and other information relevant to that function. Recall that the stack grows down, so this extra space will be at lower addresses in memory. Once the function returns, the space on the stack is freed up for future function calls. This section explains the steps of a function call in x86.

Recall that in a function call, the *caller* calls the *callee*. Program execution starts in the caller, moves to the callee as a result of the function call, and then returns to the caller after the function call completes.

When we call a function in x86, we need to update the values in all three registers we've discussed:

- eip, the instruction pointer, is currently pointing at the instructions of the caller. It needs to be changed to point to the instructions of the callee.
- ebp and esp currently point to the top and bottom of the caller stack frame, respectively. Both registers need to be updated to point to the top and bottom of a new stack frame for the callee.

When the function returns, we want to restore the old values in the registers so that we can go back to executing the caller. *When we update the value of a register, we need to save its old value on the stack so we can restore the old value after the function returns.*

There are 11 steps to calling an x86 function and returning. In this example, `main` is the caller function and `foo` is the callee function. In other words, `main` calls the `foo` function.

Here is the stack before the function is called. ebp and esp point to the top and bottom of the caller stack frame.

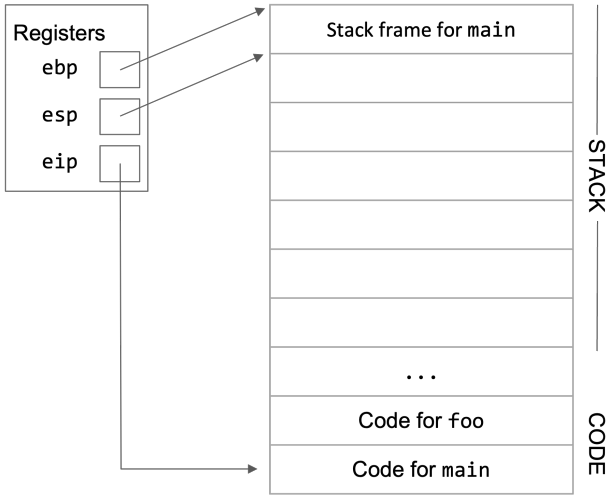

Figure 7: Initial stack diagram, with a stack frame for main at the top

1. Push arguments onto the stack. RISC-V passes arguments by storing them in registers, but x86 passes arguments by pushing them onto the stack. Note that esp is decremented as we push arguments onto the stack. Arguments are pushed onto the stack in reverse order.

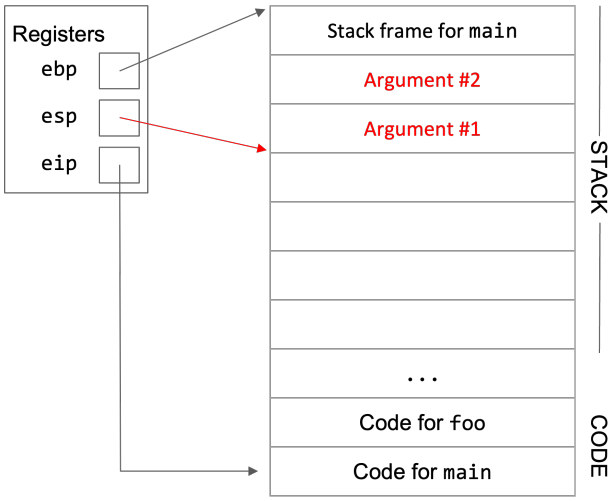

Figure 8: Next stack diagram, with argument 2 pushed below the stack frame for main and argument 1 pushed below argument 2

2. Push the old eip (rip) on the stack. We are about to change the value in the eip register, so we need to save its current value on the stack before we overwrite it with a new value. When we push this value on the stack, it is called the *old eip* or the *rip* (return instruction pointer).[6]

[6]In reality, the value we push on the stack is the current value in eip, incremented by 1 instruction. This is because after the function returns, we want to execute the instruction directly after the instruction eip is currently pointing to.

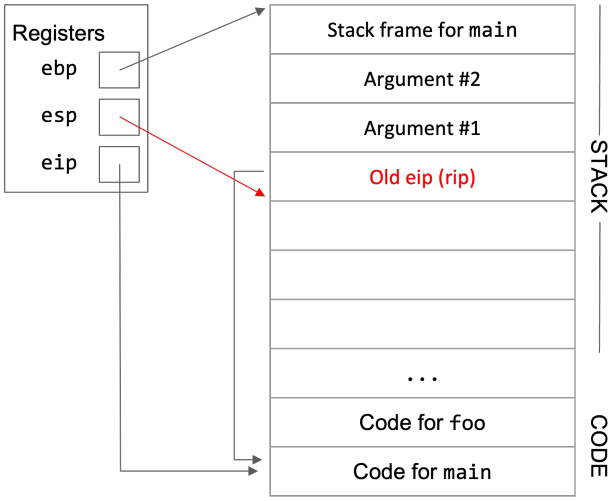

Figure 9: Next stack diagram, with the old eip pushed below argument 1

3. Move eip. Now that we've saved the old value of eip, we can safely change eip to point to the instructions for the callee function.

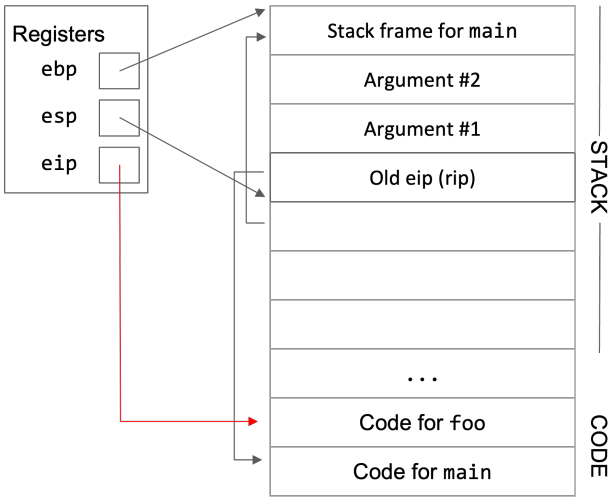

Figure 10: Next stack diagram, with the eip moved to the code for foo

4. Push the old ebp (sfp) on the stack. We are about to change the value in the ebp register, so we need to save its current value on the stack before we overwrite it with a new value. When we push this value on the stack, it is called the *old ebp* or the *sfp* (saved frame pointer). Note that esp has been decremented because we pushed a new value on the stack.

5. Move ebp down. Now that we've saved the old value of ebp, we can safely change ebp to point to the top of the new stack frame. The top of the new stack frame is where esp is currently pointing, since we are about to allocate new space below esp for the new stack frame.

6. Move esp down. Now we can allocate new space for the new stack frame by decrementing esp. The compiler looks at the complexity of the function to determine how far esp should be decremented. For example, a function with only a few local variables doesn't require too much space on the stack, so esp will only be decremented by a few bytes. On the other hand, if a function declares a large array as a local variable, esp will need to be decremented by a lot to fit the array on the stack.

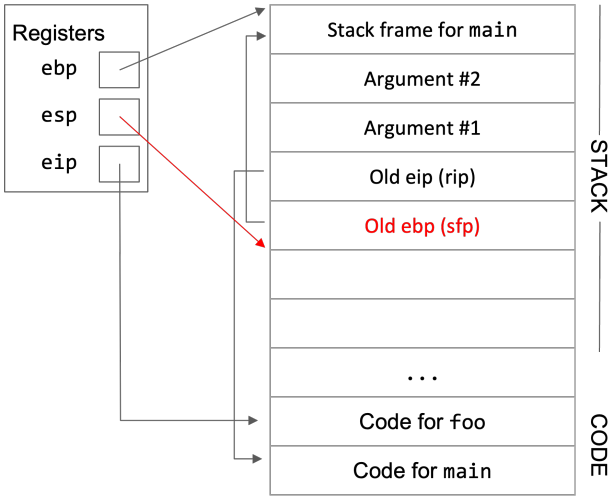

Figure 11: Next stack diagram, with the old ebp pushed below the old eip

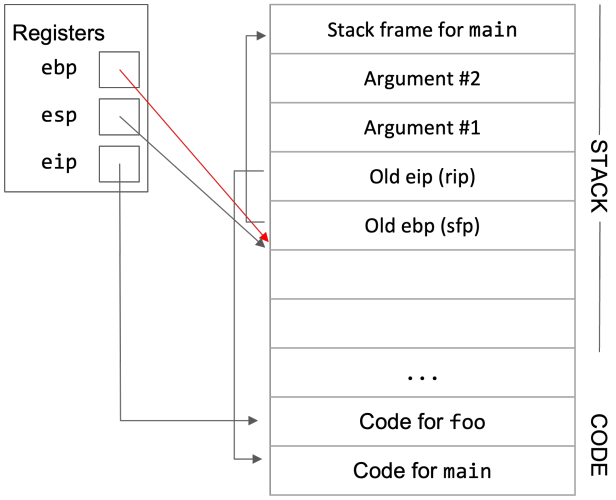

Figure 12: Next stack diagram, with the ebp moved to the esp

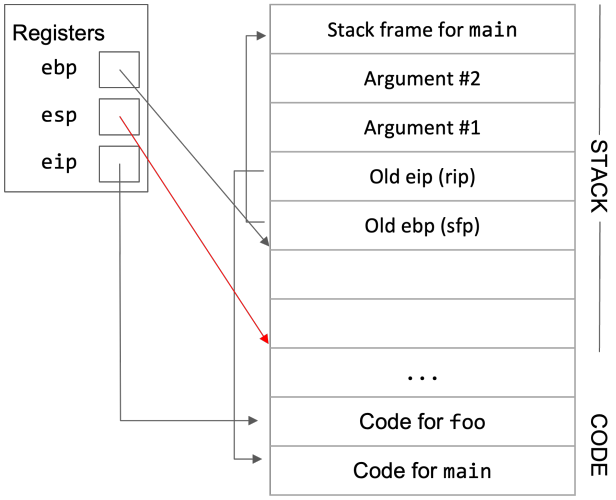

Figure 13: Next stack diagram, with the esp down by 8 bytes

7. Execute the function. Local variables and any other necessary data can now be saved in the new stack frame. Additionally, since ebp is always pointing at the top of the stack frame, we can use it as a point of reference to find other variables on the stack. For example, the arguments will be located starting at the address stored in ebp, plus 8.

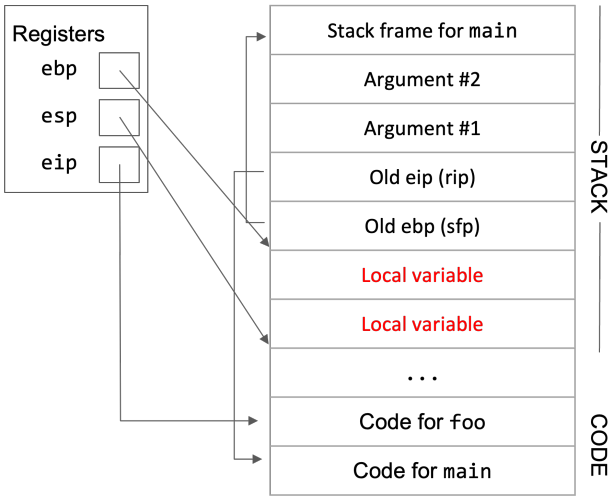

Figure 14: Next stack diagram, with the 8 bytes previously allocated now having been used for local variables

8. Move esp up. Once the function is ready to return, we increment esp to point to the top of the stack frame (ebp). This effectively erases the stack frame, since the stack frame is now located below esp. (Anything on the stack below esp is undefined.)

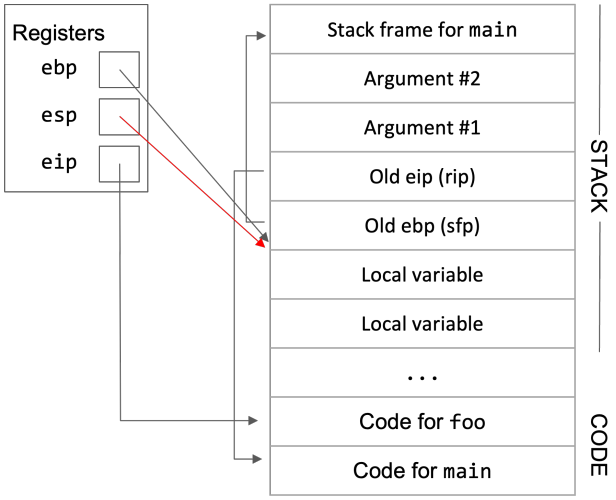

Figure 15: Next stack diagram, with the esp moved back up by 8 bytes

9. Restore the old ebp (sfp). The next value on the stack is the sfp, the old value of ebp before we started executing the function. We pop the sfp off the stack and store it back into the ebp register. This returns ebp to its old value before the function was called.

10. Restore the old eip (rip). The next value on the stack is the rip, the old value of eip before we started executing the function. We pop the rip off the stack and store it back into the eip register. This returns eip to its old value before the function was called.[7]

[7]In reality, eip is now pointing at the instruction directly after the old instruction it was pointing to. This lets us continue executing the caller function right after where we left off to call the function.

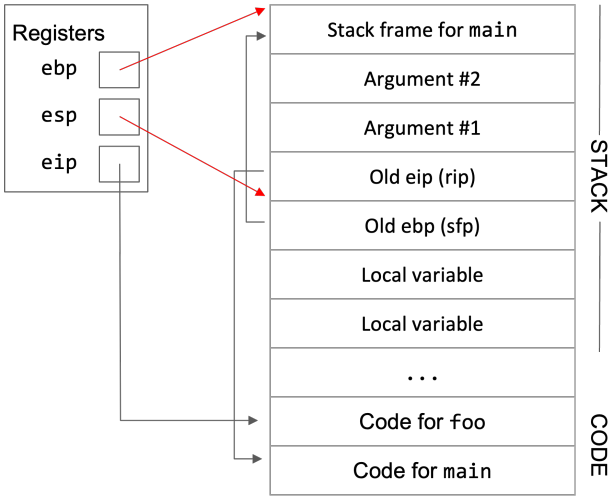

Figure 16: Next stack diagram, with the old ebp popped off the stack and the ebp moved to its location

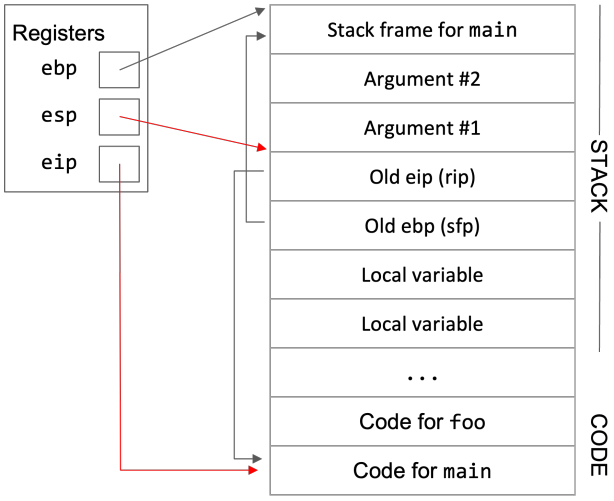

Figure 17: Next stack diagram, with the old eip popped off the stack and the eip moved to its location

11. Remove arguments from the stack. Since the function call is over, we don't need to store the arguments anymore. We can remove them by incrementing esp (recall that anything on the stack below esp is undefined).

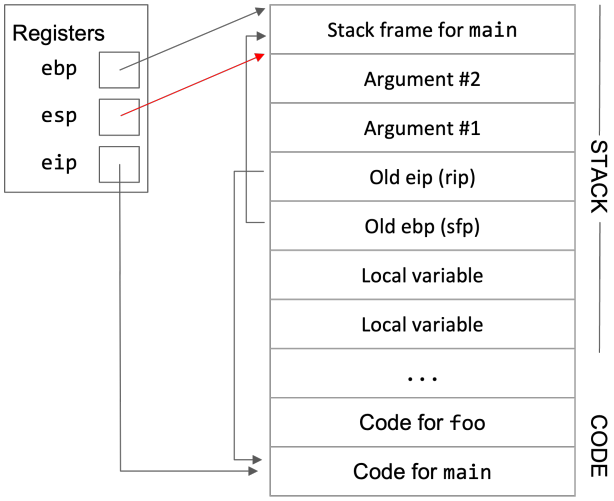

Figure 18: Next stack diagram, with the esp moved up by 8 bytes to now be above the arguments

You might notice that we saved the old values of eip and ebp during the function call, but not the old value of esp. A nice consequence of this function call design is that esp will automatically move to the bottom of the stack as we push values onto the stack and automatically return to its old position as we remove values from the stack. As a result, there is no need to save the old value of esp during the function call.

2.9. x86 function call in assembly

Consider the following C code:

```c
int main(void) {
    foo(1, 2);
}

void foo(int a, int b) {
    int bar[4];
}
```

The compiler would turn the `foo` function call into the following assembly instructions:

```
main:
    # Step 1. Push arguments on the stack in reverse order
    push $2
    push $1

    # Steps 2-3. Save old eip (rip) on the stack and change eip
    call foo

    # Execution changes to foo now. After returning from foo:

    # Step 11: Remove arguments from stack
    add $8, %esp

foo:
```

```
# Step 4. Push old ebp (sfp) on the stack
push %ebp

# Step 5. Move ebp down to esp
mov %esp, %ebp

# Step 6. Move esp down
sub $16, %esp

# Step 7. Execute the function (omitted here)

# Step 8. Move esp
mov %ebp, %esp

# Step 9. Restore old ebp (sfp)
pop %ebp

# Step 10. Restore old eip (rip)
pop %eip
```

Note that steps 1-3 happen in the caller function (`main`). Step 3 is changing the eip to point to the callee function (`foo`). Once the eip is changed, program execution is now in `foo`, where steps 4-10 take place. Step 10 is changing the eip to point back to the caller function (`main`). Once the eip is changed back, program execution is now in `main`, where step 11 takes place.

The `call` instruction in steps 2-3 pushes the old eip (rip) onto the stack and then changes eip to point to the instructions for the `foo` function.

In step 6, esp is moved down by 16 bytes. The number 16 is determined by the compiler depending on the function being called. In this case, the compiler decides 16 bytes are required to fit the local variable and any other data needed for the function to execute.

This class uses AT&T x86 syntax, which means in the `mov` instruction, the source is the first argument, and the destination is the second argument. For example, step 5, `mov %esp, %ebp` says to take the value in esp and put it in ebp.[8]

Since function calls are so common, assembly programmers sometimes use shorthand to write function returns. The two instructions in steps 8 and 9 are sometimes abbreviated as the `leave` instruction, and the instruction in step 10 is sometimes abbreviated as the `ret` instruction. This lets x86 programmers simply write "`leave ret`" after each function.

Steps 4-6 are sometimes called the *function prologue*, since they must appear at the start of the assembly code of any C function. Similarly, steps 8-10 are sometimes called the *function epilogue*.

[8]Note that if you are searching for x86 resources online, you may run into Intel syntax, where the source and destination are reversed. Percent signs `%` usually mean you're reading AT&T syntax.

3. Memory Safety Vulnerabilities

3. Memory Safety Vulnerabilities

3.1. Buffer overflow vulnerabilities

We'll start our discussion of vulnerabilities with one of the most common types of errors — *buffer overflow* (also called *buffer overrun*) vulnerabilities. Buffer overflow vulnerabilities are a particular risk in C, and since C is an especially widely used systems programming language, you might not be surprised to hear that buffer overflows are one of the most pervasive kind of implementation flaws around. However, buffer overflows are not unique to C, as C++ and Objective-C both suffer from these vulnerabilities as well.

C is a low-level language, meaning that the programmer is always exposed to the bare machine, one of the reasons why C is such a popular systems language. Furthermore, C is also a very old language, meaning that there are several legacy systems, which are old codebases written in C that are still maintained and updated. A particular weakness that we will discuss is the absence of automatic bounds-checking for array or pointer accesses. For example, if the programmer declares an array `char buffer[4]`, C will not automatically throw an error if the programmer tries to access `buffer[5]`. It is the programmer's responsibility to carefully check that every memory access is in bounds. This can get difficult as your code gets more and more complicated (e.g. for loops, user inputs, multi-threaded programs).

It is through this absence of automatic bounds-checking that buffer overflows take advantage of. A buffer overflow bug is one where the programmer fails to perform adequate bounds checks, triggering an out-of-bounds memory access that writes beyond the bounds of some memory region. Attackers can use these out-of-bounds memory accesses to corrupt the program's intended behavior.

Let us start with a simple example.

```
char buf[8];
void vulnerable() {
    gets(buf);
}
```

In this example, `gets()` reads as many bytes of input as the user supplies (through standard input), and stores them into `buf[]`. If the input contains more than 8 bytes of data, then `gets()` will write past the end of `buf`, overwriting some other part of memory. This is a bug.

Note that `char buf[8]` is defined outside of the function, so it is located in the static part of memory. Also note that each row of the diagram represents 4 bytes, so `char buf[8]` takes up 2 rows.

`gets(buf)` writes user input from lower addresses to higher addresses, starting at `buf`, and since there is no bounds checking, the attacker can overwrite parts of memory at addresses higher than `buf`.

To illustrate some of the dangers that this bug can cause, let's slightly modify the example:

```
char buf[8];
int authenticated = 0;
void vulnerable() {
    gets(buf);
}
```

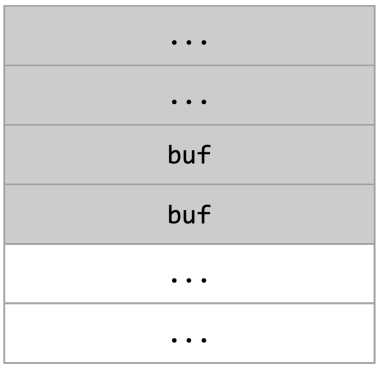

Figure 1: Two words of memory for buf overwritten and two more words of memory above it overwritten

Note that both `char buf[8]` and `authenticated` are defined outside of the function, so they are both located in the static part of memory. In C, static memory is filled in the order that variables are defined, so `authenticated` is at a higher address in memory than `buf` (since static memory grows upward and `buf` was defined first, `buf` is at a lower memory address).

Imagine that elsewhere in the code, there is a login routine that sets the `authenticated` flag only if the user proves knowledge of the password. Unfortunately, the `authenticated` flag is stored in memory right after `buf`. Note that we use "after" here to mean "at a higher memory address".

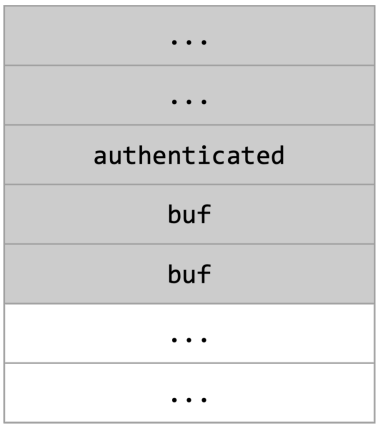

Figure 2: Two words of memory for buf overwritten and an authenticated above it overwritten

If the attacker can write 9 bytes of data to `buf` (with the 9th byte set to a non-zero value), then this will set the `authenticated` flag to true, and the attacker will be able to gain access.

The program above allows that to happen, because the `gets` function does no bounds-checking; it will write as much data to `buf` as is supplied to it by the user. In other words, the code above is *vulnerable*: an attacker who can control the input to the program can bypass the password checks.

Now consider another variation:

```
char buf[8];
int (*fnptr)();
void vulnerable() {
    gets(buf);
}
```

`fnptr` is a *function pointer*. In memory, this is a 4-byte value that stores the address of a function. In other

words, calling `fnptr` will cause the program to dereference the pointer and start executing instructions at that address.

Like `authenticated` in the previous example, `fnptr` is stored directly above `buf` in memory.

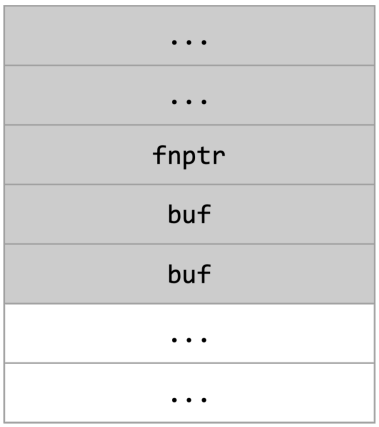

Figure 3: Two words of memory for buf overwritten and a function pointer above it overwritten

Suppose the function pointer `fnptr` is called elsewhere in the program (not shown). This enables a more serious attack: the attacker can overwrite `fnptr` with any address of their choosing, redirecting program execution to some other memory location.

Notice that in this attack, the attacker can choose to overwrite `fnptr` with any address of their choosing—so, for instance, they can choose to overwrite `fnptr` with an address where some malicious machine instructions are stored. This is a *malicious code injection* attack.

Of course, many variations on this attack are possible: the attacker could store malicious code anywhere in memory and redirect execution to that address.

Malicious code injection attacks allow an attacker to seize control of the program. At the conclusion of the attack, the program is still running, but now it is executing code chosen by the attacker, rather than the original code.

For instance, consider a web server that receives requests from clients across the network and processes them. If the web server contains a buffer overrun in the code that processes such requests, a malicious client would be able to seize control of the web server process. If the web server is running as root, once the attacker seizes control, the attacker can do anything that root can do; for instance, the attacker can leave a backdoor that allows them to log in as root later. At that point, the system has been "*owned*"[1].

The attacks illustrated above are only possible when the code satisfies certain special conditions: the buffer that can be overflowed must be followed in memory by some security-critical data (e.g., a function pointer, or a flag that has a critical influence on the subsequent flow of execution of the program). Because these conditions occur only rarely in practice, attackers have developed more effective methods of malicious code injection.

3.2. Stack smashing

One powerful method for exploiting buffer overrun vulnerabilities takes advantage of the way local variables are laid out on the stack.

Stack smashing attacks exploit the x86 function call convention. See Chapter 2 for a refresher on how x86 function calls work.

Suppose the code looks like this:

[1] You sometimes see variants on this like pwned, 0wned, ownzored, etc.

```
void vulnerable() {
    char buf[8];
    gets(buf);
}
```

When `vulnerable()` is called, a stack frame is pushed onto the stack. The stack will look something like this:

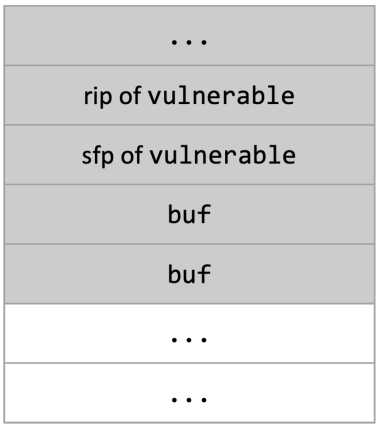

Figure 4: Two words of memory for buf overwritten and the rip and sfp above it overwritten

If the input is too long, the code will write past the end of `buf`, overwrite the sfp, and overwrite the rip. This is a *stack smashing* attack.

Note that even though we are on the stack, which "grows down," our input writes from lower addresses to higher addresses. The stack only grows down when we call a new function and need to allocate additional memory. When we call `gets`, user input is still written from lower addresses to higher addresses, just like before.

Stack smashing can be used for malicious code injection. First, the attacker arranges to inject a malicious code sequence somewhere in the program's address space, at a known address (perhaps using techniques previously mentioned). Let's suppose some malicious code exists at address `0xDEADBEEF`.

Next, the attacker provides a carefully-chosen input: `AAAAAAAAAAAA\xef\xbe\xad\xde`.

The first part of this input is a garbage byte `A` repeated many times. Since the `gets` call writes our user input starting at `buf`, we first need to overwrite all 8 bytes of `buf` with garbage. Furthermore, we don't care about the value in the sfp, so we need to overwrite the 4 bytes of the sfp with garbage. In total, we need $8 + 4 = 12$ garbage bytes at the beginning of our input.

After writing 12 garbage bytes, our next input bytes will overwrite the rip. Recall that the rip contains the address of the next instruction that will be executed after this function returns. If we overwrite the rip with some other address, then when the function returns, it will start executing instructions at that address! This is very similar to the example in the previous section, where we overwrote the function pointer with the address of malicious code.

Since malicious code exists at address `0xDEADBEEF`, the second part of our input, which overwrites the rip, is the address `0xDEADBEEF`. Note that since x86 is little-endian, we must input the bytes in reverse order: the byte `0xEF` is entered first, and the byte `0xDE` is entered last.

Now, when the `vulnerable()` function returns, the program will start executing instructions at the address in rip. Since we overwrote the rip with the address `0xDEADBEEF`, the program will start executing the malicious instructions at that address. This effectively transfers control of the program over to the attacker's malicious code.

Suppose the malicious code didn't already exist in memory, and we have to inject it ourselves during the

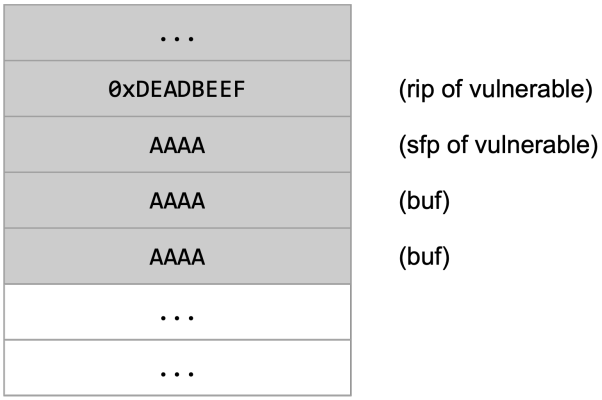

Figure 5: Two words of memory for buf and the sfp overwritten with 0xAAAA and the rip overwritten with 0xDEADBEEF

stack smashing attack. Sometimes we call this malicious code *shellcode*, because the malicious code is often written to spawn an interactive shell that lets the attacker perform arbitrary actions.

Now suppose the shellcode we want to inject is 8 bytes long. How might we place these bytes in memory? Our new input might look like this:

`[shellcode] + [4 bytes of garbage] + [address of buf]`

The first part of the input places our 8-byte shellcode at the start of the buffer.

At this point, we've entered 8 bytes, so we've filled up all of `buf`. Our next input will overwrite the sfp, but we want to overwrite the rip. As before, we will need to write some garbage bytes to overwrite the sfp so that we can overwrite the rip afterwards. We need 4 bytes of garbage to overwrite the sfp.

Finally, we overwrite the rip with the address of shellcode, as before. However, this time, the shellcode is located in the buffer, so we overwrite the rip with the address of `buf`. When the function returns, it will start executing instructions at `buf`, which causes the shellcode to execute.

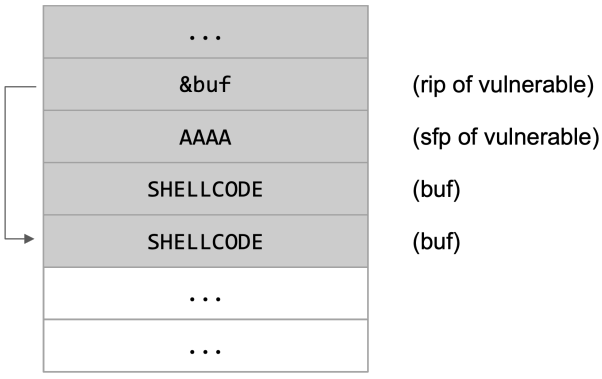

Figure 6: buf overwritten with shellcode, the sfp overwritten with 0xAAAA, and the rip overwritten with the address of buf

Now suppose our shellcode is 100 bytes long. If we try our input from before, the shellcode won't fit in the 12 bytes between the buffer and the rip. It turns out we can still craft an input to exploit the program:

`[12 bytes of garbage] + [address of rip + 4] + [shellcode]`

In this input, we place the shellcode directly above the rip in memory. The rip is 4 bytes long, so the address of the start of shellcode is 4 bytes greater than the address of the rip. When the function returns, it will start

executing instructions 4 bytes above the address of the rip, where we've placed our shellcode.

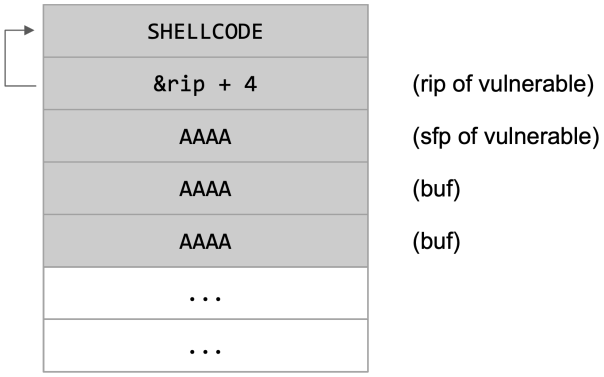

Figure 7: Two words of buf and the sfp overwritten with 0xAAAA, the rip overwritten with the address of rip + 4, and the shellcode overwritten above it

The discussion above has barely scratched the surface of techniques for exploiting buffer overrun bugs. Stack smashing dates back to at least the late 1980s, when the Morris Worm exploited a buffer overflow vulnerability to infect thousands of computers. Buffer overflows gained wider attention in 1998 with the publication of "Smashing the Stack for Fun and Profit" by Aleph One.

Modern methods are considerably more sophisticated and powerful. These attacks may seem esoteric, but attackers have become highly skilled at exploiting them. Indeed, you can find tutorials on the web explaining how to deal with complications such as:

- The malicious code is stored at an unknown location.
- The buffer is stored on the heap instead of on the stack.
- The characters that can be written to the buffer are limited (e.g., to only lowercase letters). Imagine writing a malicious sequence of instructions, where every byte in the machine code has to be in the range 0x61 to 0x7A ('a' to 'z'). Yes, it's been done.
- There is no way to introduce *any* malicious code into the program's address space.

Buffer overrun attacks may appear mysterious or complex or hard to exploit, but in reality, they are none of the above. Attackers exploit these bugs all the time. For example, the *Code Red* worm compromised 369,000 machines by exploiting a buffer overflow bug in the IIS web server. In the past, many security researchers have underestimated the opportunities for obscure and sophisticated attacks, only to later discover that the ability of attackers to find clever ways to exploit these bugs exceeded their imaginations. Attacks once thought to be esoteric to worry about are now considered easy and routinely mounted by attackers.

The bottom line is this: *If your program has a buffer overflow bug, you should assume that the bug is exploitable and an attacker can take control of your program.*

3.3. Format string vulnerabilities

Let's begin this section by walking through a normal printf call. Suppose we had the following piece of code:

```
void not_vulnerable() {
    char buf[8];
    if (fgets(buf, sizeof(buf), stdin) == NULL)
        return;
    printf(buf);
}
```

The stack diagram for this function would look something like this:

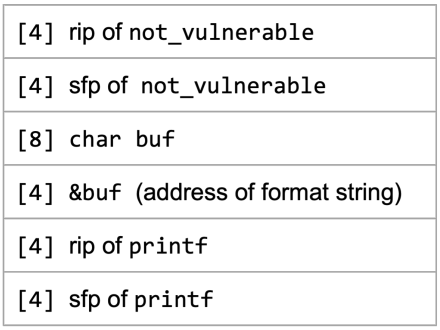

Figure 8: Initial non vulnerable code with printf

This is commonly another way we choose to represent a stack diagram, where each separate value or variable takes up one row and we dictate how many bytes it takes up in brackets to the left. For instance, the `[8] char buf` means that the `buf` character array takes up 8 bytes.

When the `printf()` function executes, it looks for a format string modifier denoted by a "%" in its first argument located 4 bytes above the RIP of `printf()`. If it finds the modifier, it then looks 8 bytes above the RIP for the "actual" argument (i.e. what the format modifier will be acting upon).

The behavior of the `printf()` function is generally controlled by the format modifier(s) that are passed into the function. The `printf()` function retrieves the parameters that are requested by the format string from the stack. Take, for example, the following line of code: `printf("x has the value %d, y has the value %d, z has the value %d \n", x, y, z);` The stack frame for this line of code would look like:

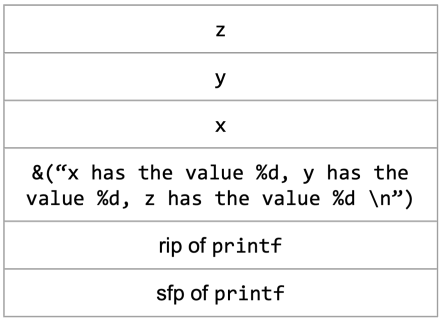

Figure 9: Not vulnerable printf statement

Remember that arguments to a function are pushed onto the stack in reverse order, which is why the address of the format string is at a lower address compared to the values of `x`, `y`, and `z`.

`printf()`'s internal pointer points to the location on the stack 8 bytes above the RIP of `printf()` due to the existence of at least one format string modifier. This internal pointer tells the function where to find the actual arguments that will be modified and eventually printed out.

A logical question you might be asking yourself might be, "Well, all this is well and good when everything works fine, but what happens when there is a mismatch in the number of format string modifiers in the first argument and number of additional arguments?" In other words, suppose our printf statement instead looked like this: `printf("x has the value %d, y has the value %d, z has the value %d \n", x, y);` Pay close attention to the fact that the format string asks for 3 arguments by having three `%d` modifiers, but we only pass in 2 arguments (i.e. `x` and `y`).

Surely the C compiler is smart enough to catch such a mistake, you might be thinking. Well, unfortunately you would be wrong. `printf()` is defined as a function with a variable number of arguments; what this means is that as long as `printf()` receives at least one argument, everything looks fine to the compiler! In

order to actually spot the mismatch, the compiler would have to understand how the `printf()` function actually works and what format string modifiers are – however, compilers aren't that sophisticated and most of them simply do not perform this kind of analysis.

Ok, well, if the C compiler doesn't catch this type of error, what about the `printf()` function itself? `printf()` simply fetches arguments from the stack according to the number of format modifiers that are present. In cases of a mismatch, it will fetch some data from the stack that does not belong to the function call.

Take the same mismatched `printf()` example we had before: `printf("x has the value %d, y has the value %d, z has the value %d \n", x, y);` The `printf()` function's internal pointer will start off 8 bytes above the RIP (since it realizes that there is at least one format modifier present). Thus, the `printf()` function takes the value 8 bytes above the RIP and prints out whatever is located there; in other words, the first `%d` consumes the value located 8 bytes above the RIP of `printf()`. Once this happens, the `printf()` function locates the next format string modifier (the second `%d`), and moves its internal pointer 4 bytes up (so now, the internal pointer is pointing 12 bytes above the RIP of `printf()`), before printing out the value located there. Finally, the `printf()` function will locate the third format string modifier, and again move its internal pointer 4 bytes up (the internal pointer is now pointing 16 bytes above the RIP of `printf()`). However, we never actually passed in a third argument to the `printf()` function, so the value located 16 bytes above the RIP of `printf()` has nothing to do with the `printf()` function at all and is instead some value left over from the previous stack frame. Since the `printf()` function does not know this, however, it looks 16 bytes above the RIP of `printf()` and prints out the value located there.

Similar to how the `%d` format modifier simply makes the `printf()` function print the value located at the expected address, various format string modifiers have different uses. Here are a couple of examples that might be useful:

- %s → Treat the argument as an address and print the string at that address up until the first null byte
- %n → Treat the argument as an address and write the number of characters that have been printed so far to that address
- %c → Treat the argument as a value and print it out as a character
- %x → Look at the stack and read the first variable after the format string
- %[b]u → Print out [b] bytes starting from the argument

The bottom line: *If your program has a format string vulnerability, assume that the attacker can learn any value stored in memory and can take control of your program.*

3.4. Integer conversion vulnerabilities

What's wrong with this code?

```
char buf[8];
void vulnerable() {
    int len = read_int_from_network();
    char *p = read_string_from_network();
    if (len > 8) {
        error("length too large: bad dog, no cookie for you!");
        return;
    }
    memcpy(buf, p, len);
}
```

Here's a hint. The function definition for `memcpy()` is:

```
void *memcpy(void *dest, const void *src, size_t n);
```

And the definition of `size_t` is:

```
typedef unsigned int size_t;
```

Do you see the bug now? If the attacker provides a negative value for `len`, the `if` statement won't notice anything wrong, and `memcpy()` will be executed with a negative third argument. C will cast this negative value to an `unsigned int` and it will become a very large positive integer. Thus `memcpy()` will copy a huge amount of memory into `buf`, overflowing the buffer.

Note that the C compiler won't warn about the type mismatch between `signed int` and `unsigned int`; it silently inserts an implicit cast. This kind of bug can be hard to spot. The above example is particularly nasty, because on the surface it appears that the programmer has applied the correct bounds checks, but they are flawed.

Here is another example. What's wrong with this code?

```
void vulnerable() {
    size_t len;
    char *buf;

    len = read_int_from_network();
    buf = malloc(len+5);
    read(fd, buf, len);
    ...
}
```

This code seems to avoid buffer overflow problems (indeed, it allocates 5 more bytes than necessary). But, there is a subtle problem: `len+5` can wrap around if `len` is too large. For instance, if `len = 0xFFFFFFFF`, then the value of `len+5` is `4` (on 32-bit platforms). In this case, the code allocates a 4-byte buffer and then writes a lot more than 4 bytes into it: a classic buffer overflow. You have to know the semantics of your programming language very well to avoid all the pitfalls.

3.5. Off-by-one vulnerabilities

Off-by-one errors are very common in programming: for example, you might accidentally use `<=` instead of `<`, or you might accidentally start a loop at `i=0` instead of `i=1`. As it turns out, even an off-by-one error can lead to dangerous memory safety vulnerabilities.

Consider a buffer whose bounds checks are off by one. This means we can write `n+1` bytes into a buffer of size `n`, overflowing the byte immediately after the buffer (but no more than that).

The following two diagrams are inspired by Section 10 of "ASLR Smack & Laugh Reference" by Tilo Müller. They show how overwriting a single byte lets you start executing instructions at an arbitrary address in memory.

Step 1: This is what normal execution during a function looks like. Consider reviewing the x86 section of the notes if you'd like a refresher. The stack has the rip (saved eip), sfp (saved ebp), and the local variable `buff`. The esp register points to the bottom of the stack. The ebp register points to the sfp at the top of the stack. The sfp (saved ebp) points to the ebp of the previous function, which is higher up in memory. The rip (saved eip) points to somewhere in the code section.

Step 2: We overwrite all of `buff`, plus the byte immediately after `buff`, which is the least significant byte of the sfp directly above `buff`. (Remember that x86 is little-endian, so the least significant byte is stored at the lowest address in memory. For example, if the sfp is `0x12345678`, we'd be overwriting the byte `0x78`.) We can change the last byte of sfp so that the sfp points to somewhere inside `buff`. The SFP label becomes FSP here to indicate that it is now a forged sfp with the last byte changed.

Eventually, after your function finishes executing, it returns. Recall from the x86 section of these notes that when a function returns, it executes the following 3 instructions:

`mov %ebp, %esp`: Change the esp register to point to wherever ebp is currently pointing.

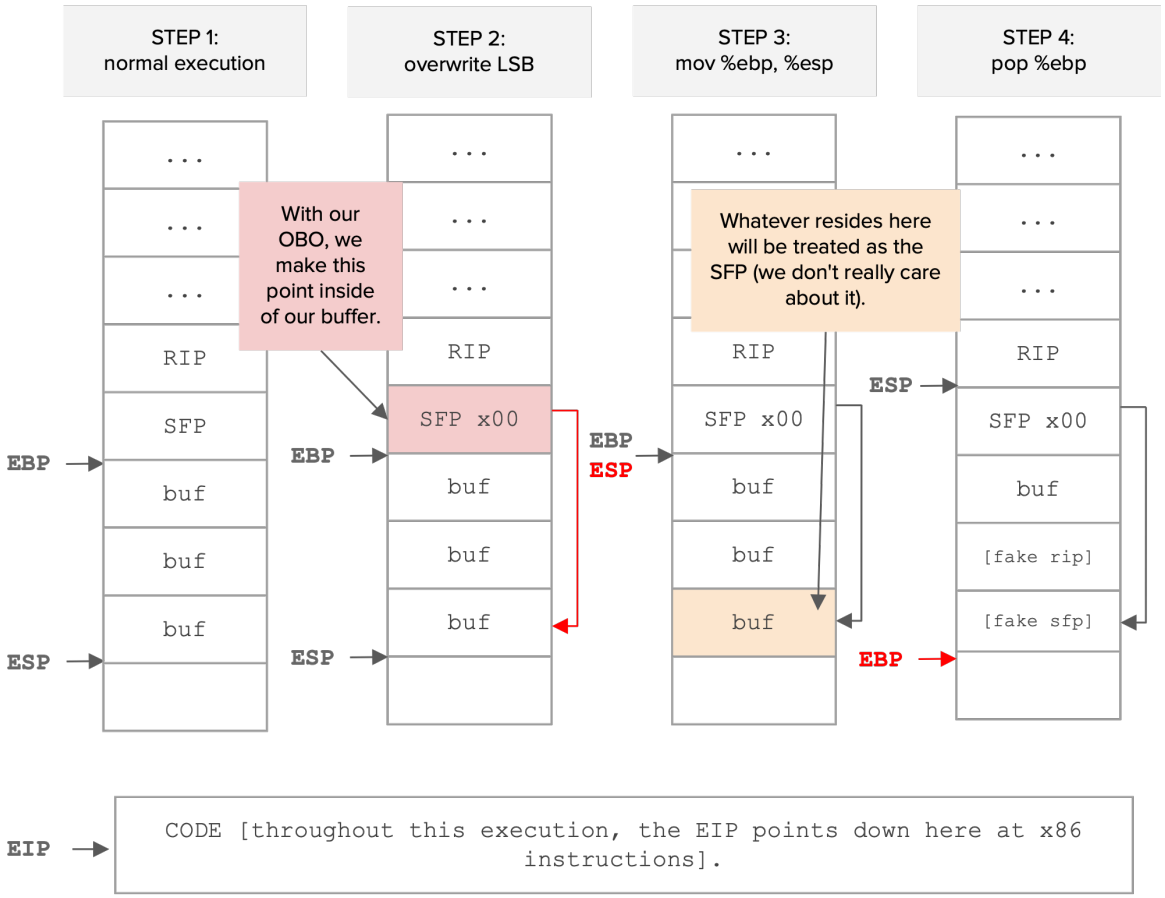

Figure 10: Stack diagrams showing the exploitation of an off-by-one vulnerability for the first return

`pop %ebp`: Take the next value on the stack (where esp is currently pointing, since esp always points to the bottom of the stack), and place it in the ebp register. Move esp up by 4 to delete this value off the stack.

`pop %eip`: Take the next value on the stack and place it in the eip register. Move esp up by 4 to "delete" this value off the stack.

In normal execution, `mov %ebp, %esp` causes esp to point to sfp (recall that ebp always points to sfp during function execution). `pop %ebp` places the next value on the stack (sfp) inside the ebp register (in other words, you're restoring the saved ebp back into ebp). `pop %eip` places the next value on the stack (rip, just above sfp) inside the eip register (in other words, you're restoring the saved eip back into eip).

So now let's see what happens if you execute these same 3 instructions when sfp incorrectly points in the buffer.

Step 3: `mov %ebp, %esp`: esp now points where ebp is pointing, which is the forged sfp.

Step 4: `pop %ebp`: Take the next value on the stack, the forged sfp, and place it in the ebp register. Now ebp is pointing inside the buffer.

Step 5: `pop %eip`: Take the next value on the stack, the rip, and place it in the eip register. Since we didn't maliciously change the rip, the old eip is correctly restored.

After step 5, nothing has changed, except that the ebp now points inside the buffer. This makes sense: we only changed the sfp (saved ebp), so when ebp is restored, it will point to where the forged sfp was pointing (inside the buffer).

The key insight for this exploit is that one function return is not enough. However, eventually, if a second function return happens, it will allow us to start executing instructions at an arbitrary location. Let's walk through the same 3 instructions again, but this time with ebp incorrectly pointing in the buffer.

Step 6: `mov %ebp, %esp`: esp now points where ebp is pointing, which is inside the buffer. At this point in normal execution, both ebp and esp think that they are pointing at the sfp.

Step 7: `pop %ebp`: Take the next value on the stack (which the program thinks is the sfp, but is actually some attacker-controlled value inside the buffer), and place it in the ebp register. The question mark here says that even though the attacker controls what gets placed in the ebp register, we don't care what the value actually is.

Step 8: `pop %eip`: Take the next value on the stack (which the program thinks is the rip, but is actually some attacker-controlled value inside the buffer), and place it in the eip register. This is where you place the address of shellcode, since you control the values in `buff`, and the program is taking an address from `buff` and jumping there to execute instructions.

In step 8, note that there is an offset of 4 from where the forged sfp points and where you should place the address of shellcode. This is because the forged sfp points to a place the program eventually tries to interpret as the sfp, but we care about the place that the program eventually tries to interpret as the rip (which is 4 bytes higher).

Also, note that it is not enough to place the shellcode 4 bytes above where the forged sfp is pointing. You need to put the address of shellcode there, since the program will interpret that part of memory as the rip.

3.6. Other memory safety vulnerabilities

Buffer overflows, format string vulnerabilities, and the other examples above are examples of *memory safety* bugs: cases where an attacker can read or write beyond the valid range of memory regions. Other examples of memory safety violations include using a dangling pointer (a pointer into a memory region that has been freed and is no longer valid) and double-free bugs (where a dynamically allocated object is explicitly freed multiple times).

"Use after free" bugs, where an object or structure in memory is deallocated (freed) but still used, are particularly attractive targets for exploitation. Exploiting these vulnerabilities generally involve the attacker

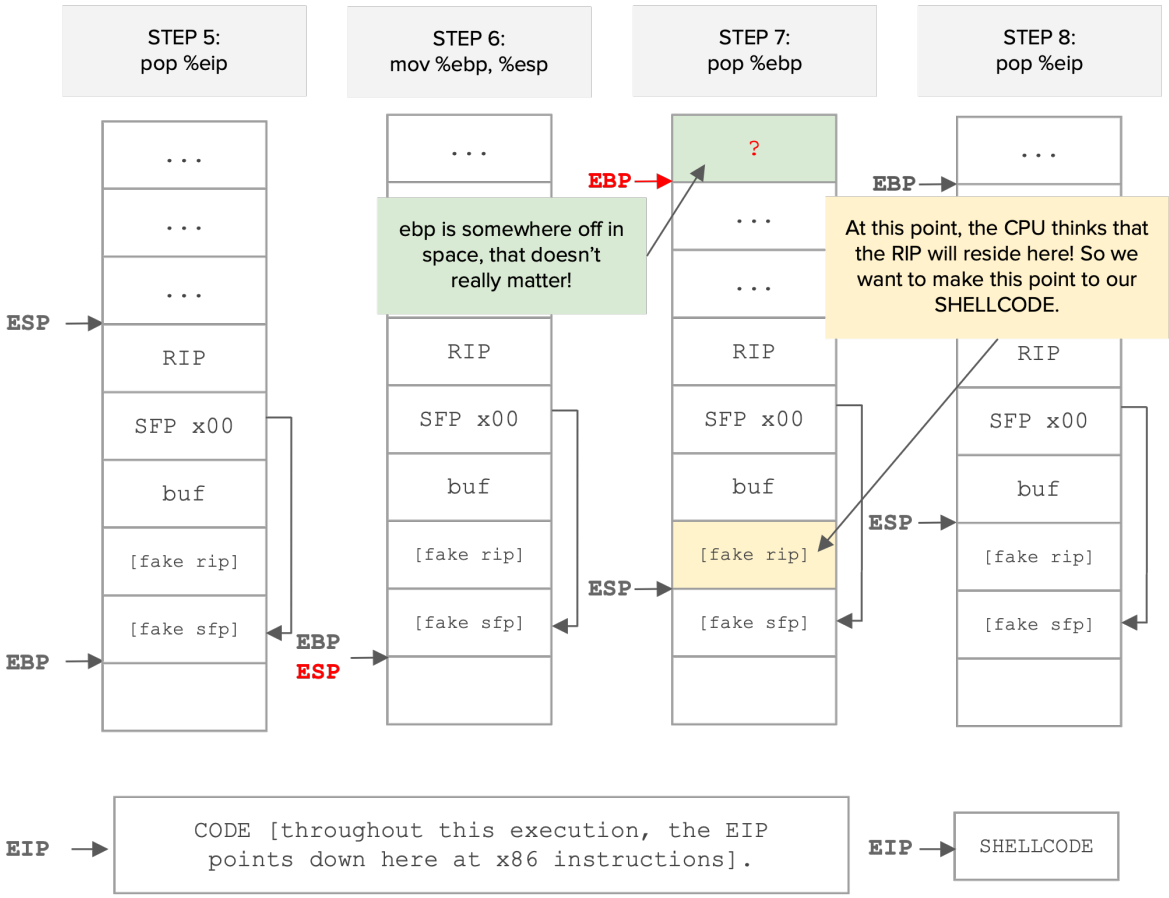

Figure 11: Stack diagrams showing the exploitation of an off-by-one vulnerability for the second return

triggering the creation of two separate objects that, because of the use-after-free on the first object, actually share the same memory. The attacker can now use the second object to manipulate the interpretation of the first object.

C++ vtable pointers are a classic example of a *heap overflow*. In C++, the programmer can declare an object on the heap. Storing an object requires storing a *vtable pointer*, a pointer to an array of pointers. Each pointer in the array contains the address of one of that object's methods. The object's instance variables are stored directly above the vtable pointer.

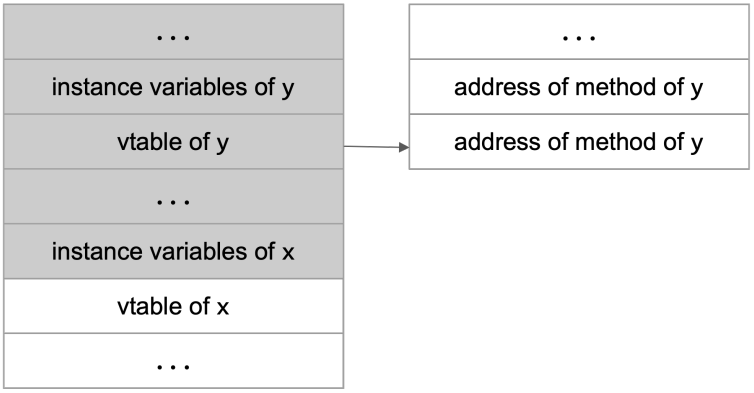

Figure 12: A diagram of a C++ object allocated in the heap, with its instance variables above its vtable

If the programmer fails to check bounds correctly, the attacker can overflow one of the instance variables of object `x`. If there is another object above `x` in memory, like object `y` in this diagram, then the attacker can overwrite that object's vtable pointer.

The attacker can overwrite the vtable pointer with the address of another attacker-controlled buffer somewhere in memory. In this buffer, the attacker can write the address of some malicious code. Now, when the program calls a method on object `y`, it will try to look up the address of the method's code in `y`'s vtable. However, `y`'s vtable pointer has been overwritten to point to attacker-controlled memory, and the attacker has written the address of some malicious code at that memory. This causes the program to start executing the attacker's malicious code.

This method of injection is very similar to stack smashing, where the attacker overwrites the rip to point to some malicious code. However, overwriting C++ vtables requires overwriting a pointer to a pointer.

4. Mitigating Memory-Safety Vulnerabilities

4. Mitigating Memory-Safety Vulnerabilities

4.1. Use a memory-safe language

Some modern languages are designed to be intrinsically memory-safe, no matter what the programmer does. Java, Python, Go, Rust, Swift, and many other programming languages include a combination of compile-time and runtime checks that prevent memory errors from occurring. Using a memory safe language is the *only* way to stop 100% of memory safety vulnerabilities. In an ideal world, everyone would program in memory-safe languages and buffer overflow vulnerabilities would no longer exist. However, because of legacy code and perceived[1] performance concerns, memory-unsafe languages such as C are still prevalent today.

4.2. Writing memory-safe code

One way to ensure memory safety is to carefully reason about memory accesses in your code, by defining pre-conditions and post-conditions for every function you write and using invariants to prove that these conditions are satisfied. Although it is a good skill to have, this process is painstakingly tedious and rarely used in practice, so it is no longer in scope for this class. If you'd like to learn more, see this lecture from David Wagner: video, slides.

Another example of defending against memory safety vulnerabilities is writing memory-safe code through defensive programming and using safe libraries. Defensive programming is very similar to defining pre and post conditions for every written function, wherein you always add checks in your code just in case something could go wrong. For example, you would always check that a pointer is not null before dereferencing it, even if you are sure that the pointer is always going to be valid. However, as mentioned earlier, this relies a lot on programmer discipline and is very tedious to properly implement. As such, a more common method is to use safe libraries, which, in turn, use functions that check bounds so you don't have to. For example, using `fgets` instead of `gets`, `strncpy` or `strlcpy` instead of `strcpy`, and `snprintf` instead of `sprintf`, are all steps towards making your code slightly more safe.

4.3. Building secure software

Yet another way to defend your code is to use tools to analyze and patch insecure code. Utilizing run-time checks that do automatic bound-checking, for example is an excellent way to help your code stay safe. If your check fails, you can direct it towards a controlled crash, ensuring that the attacker does not succeed. Hiring someone to look over your code for memory safety errors, though expensive, can prove to be extremely beneficial as well. You can also probe your own system for vulnerabilities, by subjecting your code to thorough tests. Fuzz testing, or testing with random inputs, testing corner cases, and using tools like Valgrind (to detect memory leaks), are all excellent ways to help test your code. Though it is pretty difficult to know whether you have tested your code "enough" to deem it safe, there are several code-coverage tools that can help you out.

[1] The one real performance advantage C has over a garbage collected language like Go is a far more deterministic behavior for memory allocation. But with languages like Rust, which are safe but not garbage collected, this is no longer an advantage for C.

4.4. Exploit mitigations

Sometimes you might be forced to program in a memory-unsafe language, and you cannot reason about every memory access in your code. For example, you might be asked to update an existing C codebase that is so large that you cannot go through and reason about every memory access. In these situations, a good strategy is to compile and run code with *code hardening defenses* to make common exploits more difficult.

Code hardening defenses are *mitigations*: they try to make common exploits harder and cause exploits to crash instead of succeeding, but they are not foolproof. The only way to prevent *all* memory safety exploits is to use a memory-safe language. Instead, these mitigations are best thought of as defense-in-depth: they cannot prevent all attacks, but by including many different defenses in your code, you can prevent more attacks. Over the years, there has been a back-and-forth arms race between security researchers developing new defenses and attackers developing new ways to subvert those defenses.

The rest of this section goes into more detail about some commonly-used code hardening defenses, and techniques for subverting those defenses. In many cases, using multiple mitigations produces a synergistic effect: one mitigation on its own can be bypassed, but a combination of multiple mitigations forces an attacker to discover multiple vulnerabilities in the target program.

4.5. Mitigation: Non-executable pages

Many common buffer overflow exploits involve the attacker writing some machine code into memory, and then redirecting the program to execute that injected code. For example, one of the stack smashing attacks in the previous section (`[shellcode] + [4 bytes of garbage] + [address of buf]`) involves the attacker writing machine code into memory and overwriting the rip to cause the program to execute that code.

One way to defend against this category of attacks is to make some portions of memory *non-executable*. What this means is that the computer should not interpret any data in these regions as CPU instructions. You can also think of it as not allowing the eip to ever contain the address of a non-executable part of memory.

Modern systems separate memory into *pages* in order to support virtual memory (see 61C or 162 to learn more). To defend against memory safety exploits, each page of memory is set to either be *writable or executable, but not both*. If the user can write to a page in memory, then that page of memory cannot be interpreted as machine instructions. If the program can execute a page of memory as machine instructions, then the user cannot write to that page.

This defense stops the stack smashing attack in the previous section where the attacker wrote machine code into memory. Because the attacker wrote machine code to a page in memory, that page cannot be executed as machine instructions, so the attack no longer works.

This defense has several names in practice, including W^X (Write XOR Execute), DEP (Data Execution Prevention), and the NX bit (no-execute bit).

4.6. Subverting non-executable pages: Return into libc

Non-executable pages do not stop an attacker from executing existing code in memory. Most C programs import libraries with thousands or even million lines of instructions. All of these instructions are marked as executable (and non-writable), since the programmer may want to call these functions legitimately.

An attacker can exploit this by overwriting the rip with the address of a C library function. For example, the `execv` function lets the attacker start executing the instructions of some other executable.

Some of these library functions may take arguments. For example, `execv` takes a string with the filename of the program to execute. Recall that in x86, arguments are passed on the stack. This means that an attacker can carefully place the desired arguments to the library function in the right place on the stack, so that when the library function starts to execute, it will look on the stack for arguments and find the malicious argument placed there by the attacker. The argument is not being run as code, so non-executable pages will not stop this attack.

4.7. Subverting non-executable pages: Return-oriented programming

We can take this idea of returning to already-loaded code and extend it further to now execute arbitrary code. Return-oriented programming is a technique that overwrites a chain of return addresses starting at the RIP in order to execute a series of "ROP gadgets" which are equivalent to the desired malicious code. Essentially, we are constructing a custom shellcode using pieces of code that already exist in memory. Instead of executing an existing function, like we did in "Return to libc", with ROP you can execute your own code by simply executing different pieces of different code. For example, imagine we want to add 4 to the value currently in the EDX register as part of a larger program. In loaded memory, we have the following functions:

```
foo:
    ...
    0x4005a1 <foo+33> mov %edx, %eax
    0x4005a3 <foo+35> leave
    0x4005a4 <foo+36> ret
    ...
bar:
    ...
    0x400604 <bar+20> add $0x4, %eax
    0x400608 <bar+24> pop %ebx
    0x40060a <bar+26> leave
    0x40060b <bar+27> ret
```

To emulate the `add $0x4, %edx` instruction, we could move the value in EDX to EAX using the gadget in `foo` and then add 4 to EAX using the gadget in `bar`! If we set the first return address to `0x004005a1` and the second return address to `0x00400604`, we produce the desired result. Each time we jump to ROP gadget, we eventually execute the `ret` instruction and then pop the next return address off the stack, jumping to the next gadget. We just have to keep track that our desired value is now in a different register, and because we execute a `pop %ebx` instruction in `bar` before we return, we also have to remember that the value in EBX has been updated after executing these gadgets—but these are all behaviors that we can account for using standard compiler techniques. In fact, so-called "ROP compilers" exist to take an existing vulnerable program and a desired execution flow and generate a series of return addresses.

The general strategy for executing ROPs is to write a chain of return addresses at the RIP to achieve the behavior that we want. Each return address should point to a gadget, which is a small set of assembly instructions that already exist in memory and usually end in a `ret` instruction (note that gadgets are not functions, they don't need to start with a prologue or end with an epilogue!). The gadget then executes its instructions and ends with a `ret` instruction, which tells the code to jump to the next address on the stack, thus allowing us to jump to the next gadget!

If the code base is big enough, meaning that the code imports enough libraries, there are usually enough gadgets in memory for you to be able to run any shellcode that you want. In fact, ROP compilers exist on the Internet that will automatically generate an ROP chain for you based on a target binary and desired malicious code! ROP has become so common that non-executable pages are no longer a huge issue for attackers nowadays; while having writable and executable pages makes an attacker's life easier, not a lot of effort has to be put in to subvert this defense mechanism.

4.8. Mitigation: Stack canaries

In the old days, miners would protect themselves against toxic gas buildup in the mine by bringing a caged canary into the mine. These particularly noisy birds are also sensitive to toxic gas. If toxic gas builds up in the mine, the canary dies first, which gives the miners a warning sign that the air is toxic and they should evacuate immediately. The canary in the coal mine is a sacrificial animal: the miners don't expect it to survive, but its death acts as a warning to save the lives of the miners.

We can use this same idea to prevent against buffer overflow attacks. When we call a function, the compiler places a known dummy value, the *stack canary*, on the stack. This canary value is not used by the function

at all, so it should stay unchanged throughout the duration of the function. When the function returns, the compiler checks that the canary value has not been changed. If the canary value has changed, then just like the canary in the mine dying, this is evidence that something bad has happened, and the program will crash before any further damage is done.

Like the canary in the coal mine, the stack canary is a sacrifical value: it has no purpose in the function execution and nothing bad happens if it is changed, but the canary changing acts as a warning that someone may be trying to exploit our program. This warning lets us safely crash the program instead of allowing the exploit to succeed.

The stack canary uses the fact that many common stack smashing attacks involve overflowing a local variable to overwrite the saved registers (sfp and rip) directly above. These attacks often write to *consecutive, increasing* addresses in memory, without any gaps. In other words, if the attacker starts writing at a buffer and wants to overwrite the rip, they must overwrite everything in between the buffer and the rip.

The stack canary is placed directly above the local variables and directly below the saved registers (sfp and rip):

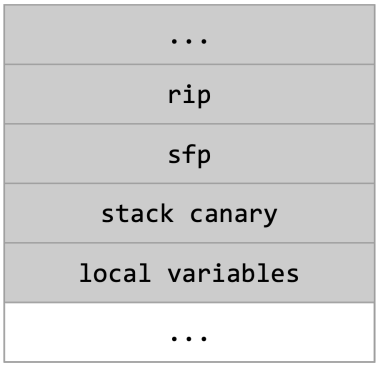

Figure 1: A stack canary located between the sfp and the local variables of a given stack frame

Suppose an attacker wants to overflow a local variable to overwrite the rip on the stack, and the vulnerability only allows the attacker to write to consecutive, increasing addresses in memory. Then the attacker must overwrite the stack canary before overwriting the rip, since the rip is located above the buffer in the stack.

Before the function returns and starts executing instructions at the rip, the compiler will check whether the canary value is unchanged. If the attacker has attempted to overwrite the rip, they will have also changed the canary value. The program will conclude that something bad is happening and crash before the attacker can take control. Note that the stack canary detects an attack before the function returns.

The stack canary is a random value generated at *runtime*. The canary is 1 word long, so it is 32 bits long in 32-bit architectures. In Project 1, the canary is 32 completely random bits. However, in reality, stack canaries are usually guaranteed to contain a null byte (usually as the first byte). This lets the canary defend against string-based memory safety exploits, such as vulnerable calls to `strcpy` that read or write values from the stack until they encounter a null byte. The null byte in the canary stops the `strcpy` call before it can copy past the canary and affect the rip.

The canary value changes each time the program is run. If the canary was the same value each time the program was run, then the attacker could run the program once, write down the canary value, then run the program again and overwrite the canary with the correct value. Within a single run of the program, the canary value is usually the same for each function on the stack.

Modern compilers automatically add stack canary checking when compiling C code. The performance overhead from checking stack canaries is negligible, and they defend against many of the most common exploits, so there is really no reason not to include stack canaries when programming in a memory-unsafe language.

4.9. Subverting stack canaries

Stack canaries make buffer overflow attacks harder for an attacker, but they do not defend programs against all buffer overflow attacks. There are many exploits that the stack canary cannot detect:

- Stack canaries can't defend against attacks outside of the stack. For example, stack canaries do nothing to protect vulnerable heap memory.
- Stack canaries don't stop an attacker from overwriting other local variables. Consider the `authenticated` example from the previous section. An attacker overflowing a buffer to overwrite the `authenticated` variable never actually changes the canary value.
- Some exploits can write to non-consecutive parts of memory. For example, format string vulnerabilities let an attacker write directly to the rip without having to overwrite everything between a local variable and the rip. This lets the attacker write "around" the canary and overwrite the rip without changing the value of the canary.

Additionally, there are several techniques for defeating the stack canary. These usually involve the attacker modifying their exploit to overwrite the canary with its original value. When the program returns, it will see that the canary is unchanged, and the program won't detect the exploit.

Guess the canary: On a 32-bit architecture, the stack canary usually only has 24 bits of entropy (randomness), because one of the four bytes is always a null byte. If the attacker runs the program with an exploit, there is a roughly 1 in 2^{24} chance that the the value the attacker is overwriting the canary with matches the actual canary value. Although the probability of success is low on one try, the attacker can simply run the program 2^{24} times and successfully exploit the program at least once with high probability.

Depending on the setting, it may be easy or hard to run a program and inject an exploit 2^{24} times. If each try takes 1 second, the attacker would need to try for over 100 days before they succeed. If the program is configured to take exponentially longer to run each time the attacker crashes it, the attacker might never be able to try enough times to succeed. However, if the attacker can try thousands of times per second, then the attacker will probably succeed in just a few hours.

On a 64-bit architecture, the stack canary has 56 bits of randomness, so it is significantly harder to guess the canary value. Even at 1,000 tries per second, an attacker would need over 2 million years on average to guess the canary!

Leak the canary: Sometimes the program has a vulnerability that allows the attacker to read parts of memory. For example, a format string vulnerability might let the attacker print out values from the stack. An attacker could use this vulnerability to leak the value of the canary, write it down, and then inject an exploit that overwrites the canary with its leaked value. All of this can happen within a single run of the program, so the canary value doesn't change on program restart.

4.10. Mitigation: Pointer authentication

As we saw earlier, stack canaries help detect if an attacker has modified the rip or sfp pointers by storing a secret value on the stack and checking if the secret value has been modified. As it turns out, we can generalize this idea of using secrets on the stack to detect when an attacker modifies *any* pointer on the stack.

Pointer authentication takes advantage of the fact that in a 64-bit architecture, many bits of the address are unused. A 64-bit address space can support 2^{64} bytes, or 18 exabytes of memory, but we are a long way off from having a machine with this much memory. A modern CPU might support a 4 terabyte address space, which means 42 bits are needed to address all of memory. This still leaves 22 unused bits in every address and pointer (the top 22 bits in the address are always 0).

Consider using these unused bits to store a secret like the stack canary. Any time we need to store an address on the stack, the CPU first replaces the 22 unused bits with some secret value, known as the *pointer authentication code*, or PAC, before pushing the value on the stack. When the CPU reads an address off the stack, it will check that the PAC is unchanged. If the PAC is unchanged, then the CPU replaces the PAC with the original unused bits and uses the address normally. However, if the PAC has been changed, this is a

warning sign that the attacker has overwritten the address! The CPU notices this and safely crashes the program.

As an example, suppose the rip of a function in a 64-bit system is `0x0000001234567899`. The address space for this architecture is 40 bits, which means the top 24 bits (3 bytes) are always 0 for every address. Instead of pushing this address directly on the stack, the CPU will first replace the 3 unused bytes with a PAC. For example, if the PAC is `0xABCDEF`, then the address pushed on the stack is `0xABCDEF1234567899`.

This address (with the secret value inserted) is invalid, and dereferencing it will cause the program to crash. When the function returns and the program needs to start executing instructions at the rip, the CPU will read this address from the stack and check that the PAC `0xABCDEF` is unchanged. If the PAC is correct, then the CPU replaces the secret with the original unused bits to make the address valid again. Now the CPU can start executing instructions at the original rip `0x0000001234567899`.

Now, an attacker trying to overwrite the rip would need to know the PAC in order to overwrite the rip with the address of some attacker shellcode. If the attacker overwrites the PAC with an incorrect value, the CPU will detect this and crash the program.

We can strengthen this defense even further. Since it is the CPU's job to add and check the PAC, we can ask the CPU to use a different PAC for every pointer stored on the stack. However, we don't want to store all these PACs on the CPU, so we'll use some special math to help us generate secure PACs on the fly.

Consider a special function $f(\mathsf{KEY}, \mathsf{ADDRESS})$. The function f takes a secret key KEY and an address ADDRESS, and outputs a PAC by performing some operation on these two inputs. This function is deterministic, which means if we supply the same key and address twice, it will output the same secret value twice. This function is also secure: an attacker who doesn't know the value of KEY cannot output secret values of their own.[2]

Now, instead of using the same PAC for every address, we can generate a different PAC for each address we store in memory. Every time an address needs to be stored in memory, the CPU runs f with the secret key and the address to generate a unique secret value. Every time an address from memory needs to be dereferenced, the CPU runs f again with the secret key and the address to re-generate the PAC, and checks that the generated value matches the value in memory. The CPU only has to remember the secret key, because all the secret values can be re-generated by running f again with the key and the address.

Using a different PAC for every address makes this defense extremely strong. An attacker who can write to random parts of memory can defeat the stack canary, but cannot easily defeat pointer authentication: they could try to leave the PAC untouched, but because they've changed the address, the old secret value will no longer check out. The CPU will run f on the attacker-generated address, and the output will be different from the old secret value (which was generated by running f on the original address). The attacker also cannot generate the correct secret value for their malicious address, because they don't know what the secret key is. Finally, an attacker could try to leak some addresses and secret values from memory, but knowing the PACs doesn't help the attacker generate a valid PAC for their chosen malicious address.

With pointer authentication enabled, an attacker is never able to overwrite pointers on the stack (including the rip) without generating the corresponding secret for the attacker's malicious address. Without knowing the key, the attacker is forced to guess the correct secret value for their address. For a 20-bit secret, the attacker has a 1 in 2^{20} chance of success.

Another way to subvert pointer authentication is to find a separate vulnerability in the program that allows the attacker to trick the program into creating a validated pointer. The attacker could also try to discover the secret key stored in the CPU, or find a way to subvert the function f used to generate the secret values.

4.11. Mitigation: Address Space Layout Randomization (ASLR)

Recall the stack smashing attacks from the previous section, where we overwrote the rip with the address of some malicious code in memory. This required knowing the exact address of the start of the malicious code.

[2]This function is called a MAC (message authentication code), and we will study it in more detail in the cryptography unit.

ASLR is a mitigation that tries to make predicting addresses in memory more difficult.

Although we showed that C memory is traditionally arranged with the code section starting at the lowest address and the stack section starting at the highest address, nothing is stopping us from shifting or rearranging the memory layout. With ASLR, each time the program is run, the beginning of each section of memory is randomly chosen. Also, if the program imports libraries, we can also randomize the starting addresses of each library's source code.

ASLR causes the absolute addresses of variables, saved registers (sfp and rip), and code instructions to be different each time the program is run. This means the attacker can no longer overwrite some part of memory (such as the rip) with a constant address. Instead, the attacker has to guess the address of their malicious instructions. Since ASLR can shuffle all four segments of memory, theoretically, certain attacks can be mitigated. By randomizing the stack, the attacker cannot place shellcode on the stack without knowing the address of the stack. By randomizing the heap, the attacker, similarly, cannot place shellcode on the heap without knowing the address of the heap. Finally, by randomizing the code, the attacker cannot construct an ROP chain or a return-to-libc attack without knowing the address of the code.

There are some constraints to randomizing the sections of memory. For example, segments usually need to start at a page boundary. In other words, the starting address of each section of memory needs to be a multiple of the page size (typically 4096 bytes in a 32-bit architecture).

Modern systems can usually implement ASLR with minimal overhead because they dynamically link libraries at runtime, which requires each segment of memory to be relocatable.

4.12. Subverting ASLR

The two main ways to subvert ASLR are similar to the main ways to subvert the stack canary: guess the address, or leak the address.

Guess the address: Because of the constraints on address randomization, a 32-bit system will sometimes only have around 16 bits of entropy for address randomization. In other words, the attacker can guess the correct address with a 1 in 2^{16} probability, or the attacker can try the exploit 2^{16} times and expect to succeed at least once. This is less of a problem on 64-bit systems, which have more entropy available for address randomization.

Like guessing the stack canary, the feasibility of guessing addresses in ASLR depends on the attack setting. For example, if each try takes 1 second, then the attacker can make 2^{16} attempts in less than a day. However, if each try after a crash takes exponentially longer, 2^{16} attempts may become infeasible.

Leak the address: Sometimes the program has a vulnerability that allows the attacker to read parts of memory. For example, a format string vulnerability might let the attacker print out values from the stack. The stack often stores absolute addresses, such as pointers and saved registers (sfp and rip). If the attacker can leak an absolute address, they may be able to determine the absolute address of other parts of memory relative to the absolute address they leaked.

Note that ASLR randomizes absolute addresses by changing the start of sections of memory, but it does not randomize the *relative* addresses of variables. For example, even if ASLR is enabled, the rip will still be 4 bytes above the sfp in a function stack frame. This means that an attacker who leaks the absolute address of the sfp could deduce the address of the rip (and possibly other values on the stack).

4.13. Combining Mitigations

We can use multiple mitigations together to force the attacker to find multiple vulnerabilities to exploit the program; this is a process known as *synergistic protection*, where one mitigation helps strengthen another mitigation. For example, combining ASLR and non-executable pages results in an attacker not being able to write their own shellcode, because of non-executable pages, and not being able to use existing code in memory, because they don't know the addresses of that code (ASLR). Thus, to defeat ASLR and non-executable pages, the attacker needs to find two vulnerabilities. First, they need to find a way to leak memory and reveal the

address location (to defeat ASLR). Next, they need to find a way to write to memory and write an ROP chain (to defeat non-executable pages).

Cryptography

Cryptography

In this unit, we'll be studying *cryptography*, techniques for securing information and communication in the presence of an attacker. In particular, we will see how we can prevent adversaries from reading or altering our private data. In a nutshell, cryptography is about communicating securely over insecure communication channels.

The ideas we'll examine have significant grounding in mathematics, and in general constitute the most systematic and formal set of approaches to security that we'll cover.

5. Introduction to Cryptography

5. Introduction to Cryptography

5.1. Disclaimer: Don't try this at home!

In this class, we will teach you the basic building blocks of cryptography, and in particular, just enough to get a feeling for how they work at a conceptual level. Understanding cryptography at a conceptual level will give you good intuition for how industrial systems use cryptography in practice.

However, cryptography in practice is very tricky to get right. Actual real-world cryptographic implementations require great attention to detail and have hundreds of possible pitfalls. For example, private information might leak out through various side-channels, random number generators might go wrong, and cryptographic primitives might lose all security if you use them the wrong way. We won't have time to teach all of those details and pitfalls to you in CS 161, so you should never implement your own cryptography using the algorithms we teach you in this class.

Instead, the cryptography we show you in this class is as much about educating you as a consumer as educating you as an engineer. If you find yourself needing an encrypted connection between two computers, or if you need to send an encrypted message to another person, you should use existing well-vetted cryptographic tools. However, you will often be faced with the problem of understanding how something is supposed to work. You might also be asked to evaluate the difference between alternatives. For that, you will need to understand the underlying cryptographic engineering involved. Similarly, there are sometimes applications that take advantage of cryptographic primitives in non-cryptographic ways, so it is useful to know the primitives. You never know when you might need a hash, an HMAC, or a block cipher for a non-security task that takes advantage of their randomness properties.

In summary, know that we're going to teach you just enough cryptography to be dangerous, but not enough to implement industrial-strength cryptography in practice.

5.2. Brief History of Cryptography

The word "cryptography" comes from the Latin roots *crypt*, meaning secret, and *graphia*, meaning writing. So cryptography is quite literally the study of how to write secret messages.

Schemes for sending secret messages go back to antiquity. 2,000 years ago, Julius Caesar employed what's today referred to as the "Caesar cypher," which consists of permuting the alphabet by shifting each letter forward by a fixed amount. For example, if Caesar used a shift by 3 then the message "cryptography" would be encoded as "fubswrjudskb". With the development of the telegraph (electronic communication) during the 1800s, the need for encryption in military and diplomatic communications became particularly important. The codes used during this "pen and ink" period were relatively simple since messages had to be decoded by hand. The codes were also not very secure, by modern standards.

The second phase of cryptography, the "mechanical era," was the result of a German project to create a mechanical device for encrypting messages in an unbreakable code. The resulting *Enigma* machine was a remarkable feat of engineering. Even more remarkable was the massive British effort during World War II to break the code. The British success in breaking the Enigma code helped influence the course of the war, shortening it by about a year, according to most experts. There were three important factors in the breaking of the Enigma code. First, the British managed to obtain a replica of a working Enigma machine from Poland,

which had cracked a simpler version of the code. Second, the Allies drew upon a great deal of brainpower, first with the Poles, who employed a large contingent of mathematicians to crack the structure, and then from the British, whose project included Alan Turing, one of the founding fathers of computer science. The third factor was the sheer scale of the code-breaking effort. The Germans figured that the Enigma was well-nigh uncrackable, but what they didn't figure on was the unprecedented level of commitment the British poured into breaking it, once codebreakers made enough initial progress to show the potential for success. At its peak, the British codebreaking organization employed over 10,000 people, a level of effort that vastly exceeded anything the Germans had anticipated. They also developed electromechanical systems that could, in parallel, search an incredible number of possible keys until the right one was found.

Modern cryptography is distinguished by its reliance on mathematics and electronic computers. It has its early roots in the work of Claude Shannon following World War II. The analysis of the *one-time pad* (discussed in the next chapter) is due to Shannon. The early 1970s saw the introduction of a standardized cryptosystem, DES, by the National Institute for Standards in Technology (NIST). DES answered the growing need for digital encryption standards in banking and other businesses. The decade starting in the late 1970s then saw an explosion of work on a computational theory of cryptography.

5.3. Definitions

Intuitively, we can see that the Caesar cypher is not secure (try all 26 possible shifts and you'll get the original message back), but how can we prove that it is, in fact, insecure? To formally study cryptography, we will have to define a mathematically rigorous framework that lets us analyze the security of various cryptographic schemes.

The rest of this section defines some important terms that will appear throughout the unit.

5.4. Definitions: Alice, Bob, Eve, and Mallory

The most basic problem in cryptography is one of ensuring the security of communications across an insecure medium. Two recurring members of the cast of characters in cryptography are *Alice* and *Bob*, who wish to communicate securely as though they were in the same room or were provided with a dedicated, untappable line. However, they only have available a telephone line or an Internet connection subject to tapping by an eavesdropping adversary, *Eve*. In some settings, Eve may be replaced by an active adversary *Mallory*, who can tamper with communications in addition to eavesdropping on them.

The goal is to design a scheme for scrambling the messages between Alice and Bob in such a way that Eve has no clue about the contents of their exchange, and Mallory is unable to tamper with the contents of their exchange without being detected. In other words, we wish to simulate the ideal communication channel using only the available insecure channel.

5.5. Definitions: Keys

The most basic building block of any cryptographic system (or *cryptosystem*) is the *key*. The key is a secret value that helps us secure messages. Many cryptographic algorithms and functions require a key as input to lock or unlock some secret value.

There are two main key models in modern cryptography. In the *symmetric key* model, Alice and Bob both know the value of a secret key, and must secure their communications using this shared secret value. In the *asymmetric key* model, each person has a secret key and a corresponding *public key*. You might remember RSA encryption from CS 70, which is an asymmetric-key encryption scheme.

5.6. Definitions: Confidentiality, Integrity, Authenticity

In cryptography, there are three main security properties that we want to achieve.

Confidentiality is the property that prevents adversaries from reading our private data. If a message is confidential, then an attacker does not know its contents. You can think about confidentiality like locking

and unlocking a message in a lockbox. Alice uses a key to lock the message in a box and then sends the message (in the locked box) over the insecure channel to Bob. Eve can see the locked box, but cannot access the message inside since she does not have a key to open the box. When Bob receives the box, he is able to unlock it using the key and retrieve the message.

Most cryptographic algorithms that guarantee confidentiality work as follows: Alice uses a key to *encrypt* a message by changing it into a scrambled form that the attacker cannot read. She then sends this encrypted message over the insecure channel to Bob. When Bob receives the encrypted message, he uses the key to *decrypt* the message by changing it back into its original form. We sometimes call the message *plaintext* when it is unencrypted and *ciphertext* when it is encrypted. Even if the attacker can see the encrypted ciphertext, they should not be able to decrypt it back into the corresponding plaintext–only the intended recipient, Bob, should be able to decrypt the message.

Integrity is the property that prevents adversaries from tampering with our private data. If a message has integrity, then an attacker cannot change its contents without being detected.

Authenticity is the property that lets us determine who created a given message. If a message has authenticity, then we can be sure that the message was written by the person who claims to have written it.

You might be thinking that authenticity and integrity seem very closely related, and you would be correct; it makes sense that before you can prove that a message came from a particular person, you first have to prove that the message was not changed. In other words, before you can prove authenticity, you first have to be able to prove integrity. However, these are not identical properties and we will take a look at some edge cases as we delve further into the cryptographic unit.

You can think about cryptographic algorithms that ensure integrity and authenticity as adding a seal on the message that is being sent. Alice uses the key to add a special seal, like a piece of tape on the envelope, on the message. She then sends the sealed message over the unsecure channel. If Mallory tampers with the message, she will break the tape on the envelope, and therefore break the seal. Without the key, Mallory cannot create her own seal. When Bob receives the message, he checks that the seal is untampered before unsealing the envelope and revealing the message.

Most cryptographic algorithms that guarantee integrity and authenticity work as follows: Alice generates a *tag* or a *signature* on a message. She sends the message with the tag to Bob. When Bob receives the message and the tag, he verifies that the tag is valid for the message that was sent. If the attacker modifies the message, the tag should no longer be valid, and Bob's verification will fail. This will let Bob detect if the message has been altered and is no longer the original message from Alice. The attacker should not be able to generate valid tags for their malicious messages.

A related property that we may want our cryptosystem to have is *deniability*. If Alice and Bob communicate securely, Alice might want to publish a message from Bob and show it to a judge, claiming that it came from Bob. If the cryptosystem has deniability, there is no cryptographic proof available to guarantee that Alice's published message came from Bob. For example, consider a case where Alice and Bob use the same key to generate a signature on a message, and Alice publishes a message with a valid signature. Then the judge cannot be sure that the message came from Bob–the signature could have plausibly been created by Alice.

5.7: Overview of schemes

We will look at cryptographic primitives that provide confidentiality, integrity, and authentication in both the symmetric-key and asymmetric-key settings.

	Symmetric-key	Asymmetric-key
Confidentiality	Block ciphers with chaining modes (e.g., AES-CBC)	Public-key encryption(e.g., El Gamal, RSA encryption)
Integrity and authentication	MACs (e.g., AES-CBC-MAC)	Digital signatures (e.g., RSA signatures)

In symmetric-key encryption, Alice uses her secret key to encrypt a message, and Bob uses the same secret key to decrypt the message.

In public-key encryption, Bob generates a matching public key and private key, and shares the public key with Alice (but does not share his private key with anyone). Alice can encrypt her message under Bob's public key, and then Bob will be able to decrypt using his private key. If these schemes are secure, then no one except Alice and Bob should be able to learn anything about the message Alice is sending.

In the symmetric-key setting, *message authentication codes (MACs)* provide integrity and authenticity. Alice uses the shared secret key to generate a MAC on her message, and Bob uses the same secret key to verify the MAC. If the MAC is valid, then Bob can be confident that no attacker modified the message, and the message actually came from Alice.

In the asymmetric-key setting, *public-key signatures* (also known as digital signatures) provide integrity and authenticity. Alice generates a matching public key and private key, and shares the public key with Bob (but does not share her private key with anyone). Alice computes a digital signature of her message using her private key, and appends the signature to her message. When Bob receives the message and its signature, he will be able to use Alice's public key to verify that no one has tampered with or modified the message, and that the message actually came from Alice.

We will also look at several other cryptographic primitives. These primitives don't guarantee confidentiality, integrity, or authenticity by themselves, but they have desirable properties that will help us build secure cryptosystems. These primitives also have some useful applications unrelated to cryptography.

- *Cryptographic hashes* provide a one way digest: They enable someone to condense a long message into a short sequence of what appear to be random bits. Cryptographic hashes are irreversible, so you can't go from the resulting hash back to the original message but you can quickly verify that a message has a given hash.
- Many cryptographic systems and problems need a lot of random bits. To generate these we use a *pseudo random number generator*, a process which takes a small amount of true randomness and stretches it into a long sequence that should be indistinguishable from actual random data.
- *Key exchange* schemes (e.g. Diffie-Hellman key exchange) allow Alice and Bob to use an insecure communication channel to agree on a shared random secret key that is subsequently used for symmetric-key encryption.

5.8. Definitions: Kerckhoff's Principle

Let's now examine the threat model, which in this setting involves answering the question: How powerful are the attackers Eve and Mallory?

To consider this question, recall *Kerckhoff's principle* from the earlier notes about security principles:

> Cryptosystems should remain secure even when the attacker knows all internal details of the system. The key should be the only thing that must be kept secret, and the system should be designed to make it easy to change keys that are leaked (or suspected to be leaked). If your secrets are leaked, it is usually a lot easier to change the key than to replace every instance of the running software. (This principle is closely related to *Shannon's Maxim: Don't rely on security through obscurity.*)

Consistent with Kerckhoff's principle, we will assume that the attacker knows the encryption and decryption algorithms.[1] The only information the attacker is missing is the secret key(s).

[1]The story of the Enigma gives one possible justification for this assumption: given how widely the Enigma was used, it was inevitable that sooner or later the Allies would get their hands on an Enigma machine, and indeed they did.

5.9. Definitions: Threat models

When analyzing the confidentiality of an encryption scheme, there are several possibilities about how much access an eavesdropping attacker Eve has to the insecure channel:

1. Eve has managed to intercept a single encrypted message and wishes to recover the plaintext (the original message). This is known as a *ciphertext-only attack*.
2. Eve has intercepted an encrypted message and also already has some partial information about the plaintext, which helps with deducing the nature of the encryption. This case is a *known plaintext attack*. In this case Eve's knowledge of the plaintext is partial, but often we instead consider complete knowledge of one instance of plaintext.
3. Eve can capture an encrypted message from Alice to Bob and re-send the encrypted message to Bob again. This is known as a *replay attack*. For example, Eve captures the encryption of the message "Hey Bob's Automatic Payment System: pay Eve 100" and sends it repeatedly to Bob so Eve gets paid multiple times. Eve might not know the decryption of the message, but she can still send the encryption repeatedly to carry out the attack.
4. Eve can trick Alice to encrypt arbitrary messages of Eve's choice, for which Eve can then observe the resulting ciphertexts. (This might happen if Eve has access to the encryption system, or can generate external events that will lead Alice to sending predictable messages in response.) At some other point in time, Alice encrypts a message that is unknown to Eve; Eve intercepts the encryption of Alice's message and aims to recover the message given what Eve has observed about previous encryptions. This case is known as a *chosen-plaintext attack*.
5. Eve can trick Bob into decrypting some ciphertexts. Eve would like to use this to learn the decryption of some other ciphertext (different from the ciphertexts Eve tricked Bob into decrypting). This case is known as a *chosen-ciphertext attack*.
6. A combination of the previous two cases: Eve can trick Alice into encrypting some messages of Eve's choosing, and can trick Bob into decrypting some ciphertexts of Eve's choosing. Eve would like to learn the decryption of some other ciphertext that was sent by Alice. (To avoid making this case trivial, Eve is not allowed to trick Bob into decrypting the ciphertext sent by Alice.) This case is known as a *chosen-plaintext/ciphertext attack*, and is the most serious threat model.

Today, we usually insist that our encryption algorithms provide security against chosen-plaintext/ciphertext attacks, both because those attacks are practical in some settings, and because it is in fact feasible to provide good security even against this very powerful attack model.

However, for simplicity, this class will focus primarily on security against chosen-plaintext attacks.

6. Symmetric-Key Cryptography

6. Symmetric-Key Encryption

In this section, we will build symmetric-key encryption schemes that guarantee confidentiality. Because we are in the symmetric key setting, in this section we can assume that Alice and Bob share a secret key that is not known to anyone else. Later we will see how Alice and Bob might securely exchange a shared secret key over an insecure communication channel, but for now you can assume that only Alice and Bob know the value of the secret key.

For modern schemes, we are going to assume that all messages are bitstrings, which is a sequence of bits, 0 or 1 (e.g. `1101100101010101`). Text, images, and most other forms of communication can usually be converted into bitstrings before encryption, so this is a useful abstraction.

6.1. IND-CPA Security

Recall from the previous chapter that confidentiality was defined to mean that an attacker cannot read our messages. This definition, while intuitive, is quite open-ended. If the attacker can read the first half of our message but not the second half, is that confidential? What if the attacker can deduce that our message starts with the words "Dear Bob?" It might also be the case that the attacker had some partial information about the message M to begin with. Perhaps she knew that the last bit of M is a 0, or that 90% of the bits of M are 1's, or that M is one of BUY! or SELL but does not know which.

A more formal, rigorous definition of confidentiality is: the ciphertext C should give the attacker no additional information about the message M. In other words, the attacker should not learn any new information about M beyond what they already knew before seeing C (seeing C should not give the attacker any new information).

We can further formalize this definition by designing an experiment to test whether the attacker has learned any additional information. Consider the following experiment: Alice has encrypted and sent one of two messages, either M_0 or M_1, and the attacker, Eve, has no idea which was sent. Eve tries to guess which was sent by looking at the ciphertext. If the encryption scheme is confidential, then Eve's probability of guessing which message was sent should be 1/2, which is the same probability as if she had not intercepted the ciphertext at all, and was instead guessing at random.

We can adapt this experiment to different threat models by allowing Eve to perform further actions as an attacker. For example, Eve might be allowed to trick Alice into encrypting some messages of Eve's choosing. Eve might also be allowed to trick Bob into decrypting some ciphertexts of Eve's choosing. In this class, we will be focusing on the chosen-plaintext attack model, which means Eve can trick Alice into encrypting some messages, but she cannot trick Alice into decrypting some messages.

In summary, our definition of confidentiality says that even if Eve can trick Alice into encrypting some messages, she still cannot distinguish whether Alice sent M_0 or M_1 in the experiment. This definition is known as indistinguishability under chosen plaintext attack, or IND-CPA. We can use an experiment or game, played between the adversary Eve and the challenger Alice, to formally prove that a given encryption scheme is IND-CPA secure or show that it is not IND-CPA secure.

The IND-CPA game works as follows:

1. The adversary Eve chooses two different messages, M_0 and M_1, and sends both messages to Alice.
2. Alice flips a fair coin. If the coin is heads, she encrypts M_0. If the coin is tails, she encrypts M_1. Formally, Alice chooses a bit $b \in \{0, 1\}$ uniformly at random, and then encrypts M_b. Alice sends the encrypted message $Enc(K, M_b)$ back to Eve.
3. Eve is now allowed to ask Alice for encryptions of messages of Eve's choosing. Eve can send a plaintext message to Alice, and Alice will always send back the encryption of the message with the secret key. Eve is allowed to repeat this as many times as she wants. Intuitively, this step is allowing Eve to perform a chosen-plaintext attack in an attempt to learn something about which message was sent.
4. After Eve is finished asking for encryptions, she must guess whether the encrypted message from step 2 is the encryption of M_0 or M_1.

If Eve can guess which message was sent with probability $> 1/2$, then Eve has won the game. This means that Eve has learned some information about which message was sent, so the scheme is not IND-CPA secure. On the other hand, if Eve cannot do any better than guess with $1/2$ probability, then Alice has won the game. Eve has learned nothing about which message was sent, so the scheme is IND-CPA secure.

There are a few important caveats to the IND-CPA game to make it a useful, practical security definition:

The messages M_0 and M_1 must be the same length. In almost all practical cryptosystems, we allow ciphertexts to leak the length of the plaintext. Why? If we want a scheme that doesn't reveal the length of the plaintext, then we would need every ciphertext to be the same length. If the ciphertext is always n bits long, then we wouldn't be able to encrypt any messages longer than n bits, which makes for a very impractical system. You could make n very large so that you can encrypt most messages, but this would mean encrypting a one-bit message requires an enormous n-bit ciphertext. Either way, such a system would be very impractical in real life, so we allow cryptosystems to leak the length of the plaintext.

If we didn't force M_0 and M_1 to be the same length, then our game would incorrectly mark some IND-CPA secure schemes as insecure. In particular, if a scheme leaks the plaintext length, it can still be considered IND-CPA secure. However, Eve would win the IND-CPA game with this scheme, since she can send a short message and a long message, see if Alice sends back a short or long ciphertext, and distinguish which message was sent. To account for the fact that cryptosystems can leak plaintext length, we use equal-length messages in the IND-CPA game.

Eve is limited to a practical number of encryption requests. In practice, some schemes may be vulnerable to attacks but considered secure anyway, because those attacks are computationally infeasible. For example, Eve could try to brute-force a 128-bit secret key, but this would take 2^{128} computations. If each computation took 1 millisecond, this would take 10^{28} years, far longer than the age of our solar system. These attacks may be theoretically possible, but they are so inefficient that we don't need to worry about attackers who try them. To account for these computationally infeasible attacks in the IND-CPA game, we limit Eve to a practical number of encryption requests. One commonly-used measure of practicality is polynomially-bounded runtime: any algorithm Eve uses during the game must run in $O(n^k)$ time, for some constant k.

Eve only wins if she has a non-negligible advantage. Consider a scheme where Eve can correctly which message was sent with probability $1/2 + 1/2^{128}$. This number is greater than $1/2$, but Eve's advantage is $1/2^{128}$, which is astronomically small. In this case, we say that Eve has *negligible* advantage–the advantage is so small that Eve cannot use it to mount any practical attacks. For example, the scheme might use a 128-bit key, and Eve can break the scheme if she guesses the key (with probability $1/2^{128}$). Although this is theoretically a valid attack, the odds of guessing a 128-bit key are so astronomically small that we don't need to worry about it. The exact definition of negligible is beyond the scope of this class, but in short, Eve only wins the IND-CPA game if she can guess which message was sent with probability greater than $1/2 + n$, where n is some non-negligible probability.

You might have noticed that in step 3, there is nothing preventing Eve from asking Alice for the encryption of M_0 or M_1 again. This is by design: it means any deterministic scheme is not IND-CPA secure, and it forces any IND-CPA secure scheme to be non-deterministic. Informally, a deterministic scheme is one that, given a particular input, will always produce the same output. For example, the Caesar Cipher that was seen

in the previous chapter is a deterministic scheme since giving it the same input twice will always produce the same output (i.e. inputting "abcd" will always output "cdef" when we shift by 2). As we'll see later, deterministic schemes do leak information, so this game will correctly classify them as IND-CPA insecure. In a later section we'll also see how to win the IND-CPA game against a deterministic scheme.

6.2. XOR review

Symmetric-key encryption often relies on the bitwise XOR (exclusive-or) operation (written as $\oplus$), so let's review the definition of XOR.

$0 \oplus 0 = 0$

$0 \oplus 1 = 1$

$1 \oplus 0 = 1$

$1 \oplus 1 = 0$

Given this definition, we can derive some useful properties:

$x \oplus 0 = x$	0 is the identity
$x \oplus x = 0$	x is its own inverse
$x \oplus y = y \oplus x$	commutative property
$(x \oplus y) \oplus z = x \oplus (y \oplus z)$	associative property

One handy identity that follows from these is: $x \oplus y \oplus x = y$. In other words, given $(x \oplus y)$, you can retrieve y by computing $(x \oplus y) \oplus x$, effectively "cancelling out" the x.

We can also perform algebra with the XOR operation:

$y \oplus 1 = 0$	goal: solve for y
$y \oplus 1 \oplus 1 = 0 \oplus 1$	XOR both sides by 1
$y = 1$	simplify left-hand side using the identity above

6.3. One Time Pad

The first symmetric encryption scheme we'll look at is the *one-time pad (OTP)*. The one time pad is a simple and idealized encryption scheme that helps illustrate some important concepts, though as we will see shortly, it is impractical for real-world use.

In the one-time pad scheme, Alice and Bob share an n-bit secret key $K = k_1 \cdots k_n$ where the bits $k_1, \ldots k_n$ are picked uniformly at random (they are the outcomes of independent unbiased coin flips, meaning that to pick k_1 a coin is flipped and if it lands on heads, then k_1 is assigned 1, but if it lands on tails, k_1 is assigned 0).

Suppose Alice wishes to send the n-bit message $M = m_1 \cdots m_n$.

The desired properties of the encryption scheme are:

1. It should scramble up the message, i.e., map it to a ciphertext $C = c_1 \cdots c_n$.
2. Given knowledge of the secret key K, it should be easy to recover M from C.
3. Eve, who does not know K, should get *no* information about M.

Encryption in the one-time pad is very simple: $c_j = m_j \oplus k_j$. In words, you perform a bitwise XOR of the message and the key. The jth bit of the ciphertext is the jth bit of the message, XOR with the jth bit of the key.

We can derive the decryption algorithm by doing some algebra on the encryption equation:

$c_j = m_j \oplus k_j$	encryption equation, solve for m_j
$c_j \oplus k_j = m_j \oplus k_j \oplus k_j$	XOR both sides by k_j
$c_j \oplus k_j = m_j$	simplify right-hand side using the handy identity from above

In words, given ciphertext C and key K, the jth bit of the plaintext is the jth bit of the ciphertext, XOR with the jth bit of the key.

To sum up, the one-time pad is described by specifying three procedures:

- Key generation: Alice and Bob pick a shared random key K.
- Encryption algorithm: $C = M \oplus K$.
- Decryption algorithm: $M = C \oplus K$.

One-time pad is information-theoretically secure, in the sense that a ciphertext leaks precisely zero information about its plaintext. For a fixed choice of plaintext M, every possible value of the ciphertext C can be achieved by an appropriate and unique choice of the shared key K: namely $K = M \oplus C$. Since each such key value K is equally likely, it follows that C is also equally likely to be any n-bit string. Thus the eavesdropper sees a uniformly random n bit string no matter what the plaintext message was, presuming the key is randomly chosen and used only once.

The one time pad has a major drawback. As its name suggests, the shared key cannot be reused to transmit another message M'. If the key K is reused to encrypt two messages M and M', then Eve can take the XOR of the two ciphertexts $C = M \oplus K$ and $C' = M' \oplus K$ to obtain $C \oplus C' = M \oplus M'$. This gives partial information about the two messages. In particular, if Eve happens to learn M, then she can deduce the other message M'. In other words, given $M \oplus M'$ and M, she can calculate $M' = (M \oplus M') \oplus M$. Actually, in this case, she can reconstruct the key K, too. Question: How?[1]

In practice, even if Eve does not know M or M', often there is enough redundancy in messages that merely knowing $M \oplus M'$ is enough to recover most of M and M'. For instance, the US exploited this weakness to read some World War II era Soviet communications encrypted with the one-time pad, when US cryptanalysts discovered that Soviet officials in charge of generating random keys for the one-time pad got lazy and started re-using old keys. The VENONA project, although initiated just shortly after World War II, remained secret until the early 1980s.

We can see that the one-time pad with key reuse is insecure because Eve has learned something about the original messages (namely, the XOR of the two original messages). We can also formally prove that the one-time pad with key reuse is not IND-CPA secure by showing a strategy for the adversary Eve to correctly guess which message was encrypted, with probability greater than 1/2.

Eve sends two messages, M_0 and M_1 to the challenger. The challenger randomly chooses one message to encrypt and sends it back to Eve. At this point, Eve knows she has received either $M_0 \oplus K$ or $M_1 \oplus K$, depending on which message was encrypted. Eve is now allowed to ask for the encryption of arbitrary messages, so she queries the challenger for the encryption of M_0. The challenger is using the same key for every message, so Eve will receive $M_0 \oplus K$. Eve can now compare this value to the encryption she is trying to guess: if the value matches, then Eve knows that the challenger encrypted M_0 and sent $M_0 \oplus K$. If the value doesn't match, then Eve knows that the challenger encrypted M_1 and sent $M_1 \oplus K$. Thus Eve can guess which message the challenger encrypted with 100% probability! This is greater than 1/2 probability, so Eve has won the IND-CPA game, and we have proven that the one-time pad scheme with key reuse is insecure.

Consequently, the one-time pad is not secure if the key is used to encrypt more than one message. This makes it impractical for almost all real-world situations–if Alice and Bob want to encrypt an n-bit message with a one-time pad, they will first need to securely send each other a new, previously unused n-bit key. But if they've found a method to securely exchange an n-bit key, they could have just used that same method to exchange the n-bit message![2]

[1] Answer: Given M and $C = M \oplus K$, Eve can calculate $K = M \oplus C$.

[2] This is why the only primary users of one-time-pads are spies in the field. Before the spy leaves, they obtain a large amount of key material. Unlike the other encryption systems we'll see in these notes, a one-time pad can be processed entirely with

6.4. Block Ciphers

As we've just seen, generating new keys for every encryption is difficult and expensive. Instead, in most symmetric encryption schemes, Alice and Bob share a secret key and use this single key to repeatedly encrypt and decrypt messages. The block cipher is a fundamental building block in implementing such a symmetric encryption scheme.

Intuitively, a block cipher transforms a fixed-length, n-bit input into a fixed-length n-bit output. The block cipher has 2^k different settings for scrambling, so it also takes in a k-bit key as input to determine which scrambling setting should be used. Each key corresponds to a different scrambling setting. The idea is that an attacker who doesn't know the secret key won't know what mode of scrambling is being used, and thus won't be able to decrypt messages encrypted with the block cipher.

A block cipher has two operations: encryption takes in an n-bit plaintext and a k-bit key as input and outputs an n-bit ciphertext. Decryption takes in an n-bit ciphertext and a k-bit key as input and outputs an n-bit plaintext. Question: why does the decryption require the key as input?[3]

Given a fixed scrambling setting (key), the block cipher encryption must map each of the 2^n possible plaintext inputs to a different ciphertext output. In other words, given a specific key, the block cipher encryption must be able to map every possible input to a unique output. If the block cipher mapped two plaintext inputs to the same ciphertext output, there would be no way to decrypt that ciphertext back into plaintext, since that ciphertext could correspond to multiple different plaintexts. This means that the block cipher must also be *deterministic*. Given the same input and key, the block cipher should always give the same output.

In mathematical notation, the block cipher can be described as follows. There is an encryption function $E : \{0,1\}^k \times \{0,1\}^n \to \{0,1\}^n$. This notation means we are mapping a k-bit input (the key) and an n-bit input (the plaintext message) to an n-bit output (the ciphertext). Once we fix the key K, we get a function mapping n bits to n bits: $E_K : \{0,1\}^n \to \{0,1\}^n$ defined by $E_K(M) = E(K, M)$. E_K is required to be a *permutation* on the n-bit strings, in other words, it must be an invertible (bijective) function. The inverse mapping of this permutation is the decryption algorithm D_K. In other words, decryption is the reverse of encryption: $D_K(E_K(M)) = M$.

The block cipher as defined above is a category of functions, meaning that there are many different implementations of a block cipher. Today, the most commonly used block cipher implementation is called Advanced Encryption Standard (AES). It was designed in 1998 by Joan Daemen and Vincent Rijmen, two researchers from Belgium, in response to a competition organized by NIST.[4]

AES uses a block length of $n = 128$ bits and a key length of $k = 128$ bits. It can also support $k = 192$ or $k = 256$ bit keys, but we will assume 128-bit keys in this class. It was designed to be extremely fast in both hardware and software.

6.5. Block Cipher Security

Block ciphers, including AES, are not IND-CPA secure on their own because they are deterministic. In other words, encrypting the same message twice with the same key produces the same output twice. The strategy that an adversary, Eve, uses to break the security of AES is exactly the same as the strategy from the one-time pad with key reuse. Eve sends M_0 and M_1 to the challenger and receives either $E(K, M_0)$ or $E(K, M_1)$. She then queries the challenger for the encryption of M_0 and receives $E(K, M_0)$. If the two encryptions she receives from the challenger are the same, then Eve knows the challenger encrypted M_0 and sent $E(K, M_0)$. If the two encryptions are different, then Eve knows the challenger encrypted M_1 and sent

pencil and paper. The spy then broadcasts messages encrypted with the one-time pad to send back to their home base. To obfuscate the spy's communication, there are also "numbers stations" that continually broadcast meaningless sequences of random numbers. Since the one-time pad is IND-CPA secure, an adversary can't distinguish between the random number broadcasts and the messages encoded with a one time pad.

[3] Answer: The key is needed to determine which scrambling setting was used to generate the ciphertext. If decryption didn't require a key, any attacker would be able to decrypt encrypted messages!

[4] Fun fact: Professor David Wagner, who sometimes teaches this class, was part of the team that came up with a block cipher called TwoFish, which was one of the finalists in the NIST competition.

$E(K, M_1)$. Thus Eve can win the IND-CPA game with probability $100\% > 1/2$, and the block cipher is not IND-CPA secure.

Although block ciphers are not IND-CPA secure, they have a desirable security property that will help us build IND-CPA secure symmetric encryption schemes: namely, a block cipher is *computationally indistinguishable* from a random permutation. In other words, for a fixed key K, E_K "behaves like" a random permutation on the n-bit strings.

A random permutation is a function that maps each n-bit input to exactly one random n-bit output. One way to generate a random permutation is to write out all 2^n possible inputs in one column and all 2^n possible outputs in another column, and then draw 2^n random lines connecting each input to each output. Once generated, the function itself is not random: given the same input twice, the function gives the same output twice. However, the choice of which output is given is randomly determined when the function is created.

Formally, we perform the following experiment to show that a block cipher is indistinguishable from a random permutation. The adversary, Eve, is given a box which contains either (I) the encryption function E_K with a randomly chosen key K, or (II) a permutation π on n bits chosen uniformly at random when the box was created (in other words, map each n-bit input to a different random n-bit output). The type of box given to Eve is randomly selected, but we don't tell Eve which type of box she has been given. We also don't tell Eve the value of the key K.

Eve is now allowed to play with the box as follows: Eve can supply an input x to the box and receive a corresponding output y from the box (namely, $y = E_K(x)$ for a type-I box, or $y = \pi(x)$ for a type-II box). After playing with the box, Eve must guess whether the box is type I or type II. If the block cipher is truly indistinguishable from random, then Eve cannot guess which type of box she received with probability greater than $1/2$.

AES is not truly indistinguishable from random, but it is believed to be *computationally* indistinguishable from random. Intuitively, this means that given a practical amount of computation power (e.g. polynomially-bounded runtime), Eve cannot guess which type of box she received with probability greater than $1/2$. Another way to think of computational indistinguishability is: Eve can guess which type of box she received with probability $1/2$, plus some negligible amount (e.g. $1/2^{128}$). With infinite computational time and power, Eve could leverage this tiny $1/2^{128}$ advantage to guess which box she received, but with only a practical amount of computation power, this advantage is useless for Eve.

The computational indistinguishability property of AES gives us a strong security guarantee: given a single ciphertext $C = E_K(M)$, an attacker without the key cannot learn anything about the original message M. If the attacker could learn something about M, then AES would no longer be computationally indistinguishable: in the experiment from before, Eve could feed M into the box and see if given only the output from the box, she can learn something about M. If Eve learns something about M, then she knows the output came from a block cipher. If Eve learns nothing about M, then she knows the output came from a random permutation. However, since we believe that AES is computationally indistinguishable from random, we can say that an attacker who receives a ciphertext learns *nothing* about the original message.

There is no proof that AES is computationally indistinguishable from random, but it is believed to be computationally indistinguishable. After all these years, the best known attack is still *exhaustive key search*, where the attacker systematically tries decrypting some ciphertext using every possible key to see which one gives intelligible plaintext. Given infinite computational time and power, exhaustive key search can break AES, which is why it is not truly indistinguishable from random. However, with a 128-bit key, exhaustive key search requires 2^{128} computations in the worst case (2^{127} on average). This is a large enough number that even the fastest current supercomputers couldn't possibly mount an exhaustive key search attack against AES within the lifetime of our Solar system.

Thus AES behaves very differently than the one-time pad. Even given a very large number of plaintext/ciphertext pairs, there appears to be no effective way to decrypt any new ciphertexts. We can leverage this property to build symmetric-key encryption schemes where there is no effective way to decrypt *any* ciphertext, even if it's the encryption of a message we've seen before.

6.6. Block Cipher Modes of Operation

There are two main reasons AES by itself cannot be a practical IND-CPA secure encryption scheme. The first is that we'd like to encrypt arbitrarily long messages, but the block cipher only takes fixed-length inputs. The other is that if the same message is sent twice, the ciphertext in the two transmissions is the same with AES (i.e. it is deterministic). To fix these problems, the encryption algorithm can either be randomized or stateful—it either flips coins during its execution, or its operation depends upon some state information. The decryption algorithm, however, is neither randomized nor stateful.

There are several standard ways (or modes of operation) of building an encryption algorithm, using a block cipher:

ECB Mode (Electronic Code Book): In this mode the plaintext M is simply broken into n-bit blocks $M_1 \cdots M_l$, and each block is encoded using the block cipher: $C_i = E_K(M_i)$. The ciphertext is just a concatenation of these individual blocks: $C = C_1 \cdot C_2 \cdots C_l$. This scheme is **flawed**. Any redundancy in the blocks will show through and allow the eavesdropper to deduce information about the plaintext. For instance, if $M_i = M_j$, then we will have $C_i = C_j$, which is visible to the eavesdropper; so ECB mode **leaks information** about the plaintext.

- ECB mode encryption: $C_i = E_K(M_i)$
- ECB mode decryption: $M_i = D_K(C_i)$

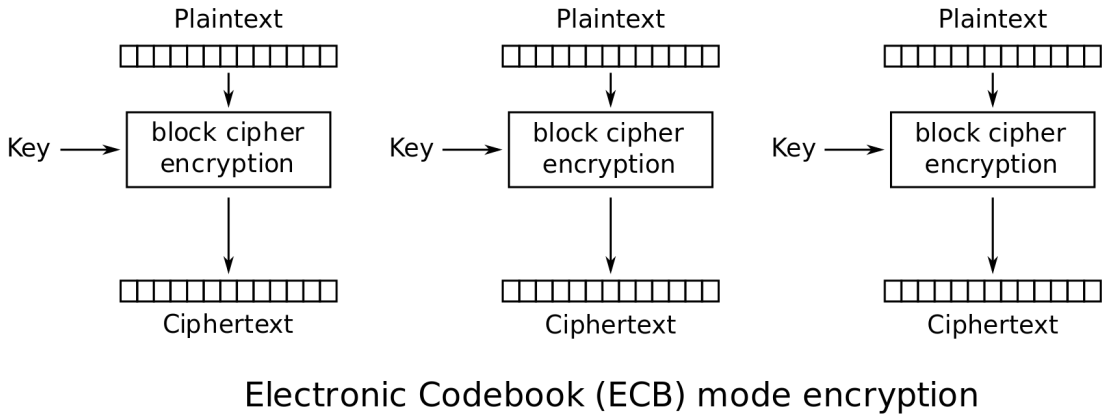

Figure 1: Diagram of encryption for the ECB mode of operation

CBC Mode (Cipher Block Chaining): This is a popular mode for commercial applications. For each message the sender picks a random n-bit string, called the *initial vector* or IV. Define $C_0 = IV$. The i^{th} ciphertext block is given by $C_i = E_K(C_{i-1} \oplus M_i)$. The ciphertext is the concatenation of the initial vector and these individual blocks: $C = IV \cdot C_1 \cdot C_2 \cdots C_l$. CBC mode has been proven to provide strong security guarantees on the privacy of the plaintext message (assuming the underlying block cipher is secure).

- CBC mode encryption: $\begin{cases} C_0 = IV \\ C_i = E_K(P_i \oplus C_{i-1}) \end{cases}$
- CBC mode decryption: $P_i = D_K(C_i) \oplus C_{i-1}$

CFB Mode (Ciphertext Feedback Mode): This is another popular mode with properties very similar to CBC mode. Again, C_0 is the IV. The i^{th} ciphertext block is given by $C_i = E_K(C_{i-1}) \oplus M_i$.

- CFB mode encryption:

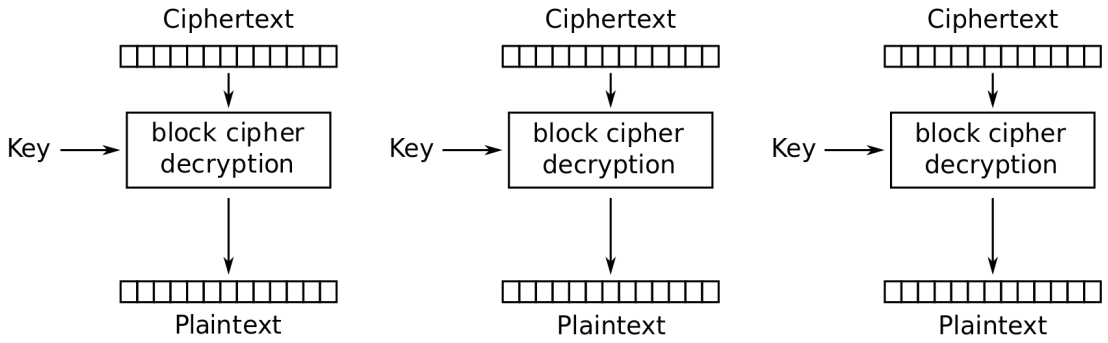

Figure 2: Diagram of decryption for the ECB mode of operation

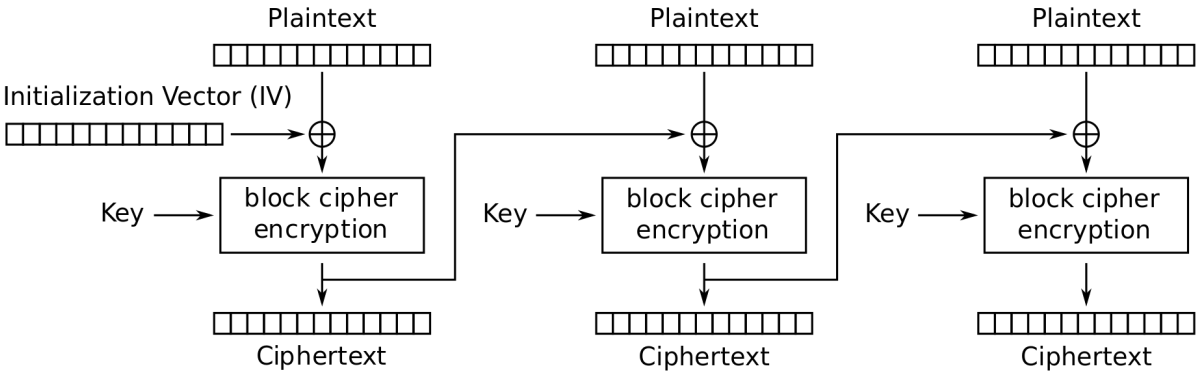

Figure 3: Diagram of encryption for the CBC mode of operation

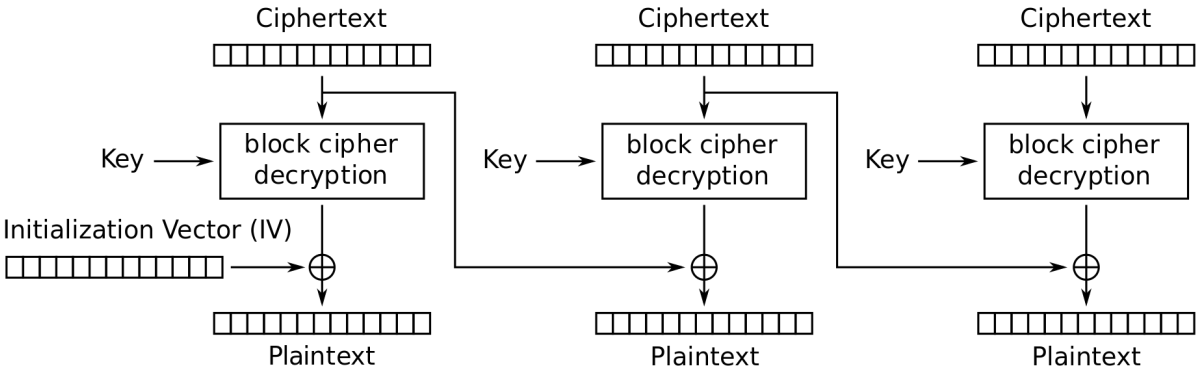

Figure 4: Diagram of decryption for the CBC mode of operation

$$\begin{cases} C_0 = IV \\ C_i = E_K(C_{i-1}) \oplus P_i \end{cases}$$

- CFB mode decryption: $P_i = E_K(C_{i-1}) \oplus C_i$

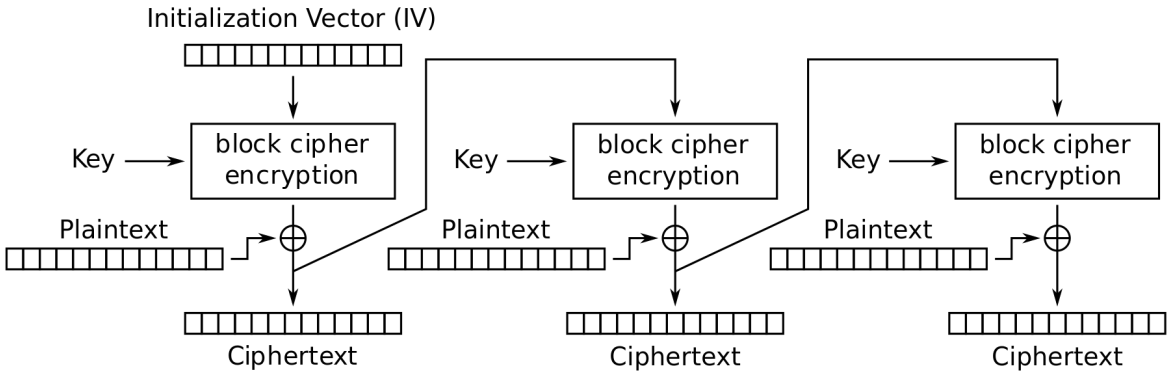

Figure 5: Diagram of encryption for the CFB mode of operation

OFB Mode (Output Feedback Mode): In this mode, the initial vector IV is repeatedly encrypted to obtain a set of values Z_i as follows: $Z_0 = IV$ and $Z_i = E_K(Z_{i-1})$. These values Z_i are now used as though they were the key for a one-time pad, so that $C_i = Z_i \oplus M_i$. The ciphertext is the concatenation of the initial vector and these individual blocks: $C = IV \cdot C_1 \cdot C_2 \cdots C_l$. In OFB mode, it is very easy to tamper with ciphertexts. For instance, suppose that the adversary happens to know that the j^{th} block of the message, M_j, specifies the amount of money being transferred to his account from the bank, and suppose he also knows that $M_j = 100$. Since he knows both M_j and C_j, he can determine Z_j. He can then substitute any n-bit block in place of M_j and get a new ciphertext C'_j where the 100 is replaced by any amount of his choice. This kind of tampering is also possible with other modes of operation as well (so don't be fooled into thinking that CBC mode is safe from tampering); it's just easier to illustrate on OFB mode.

- OFB mode encryption:

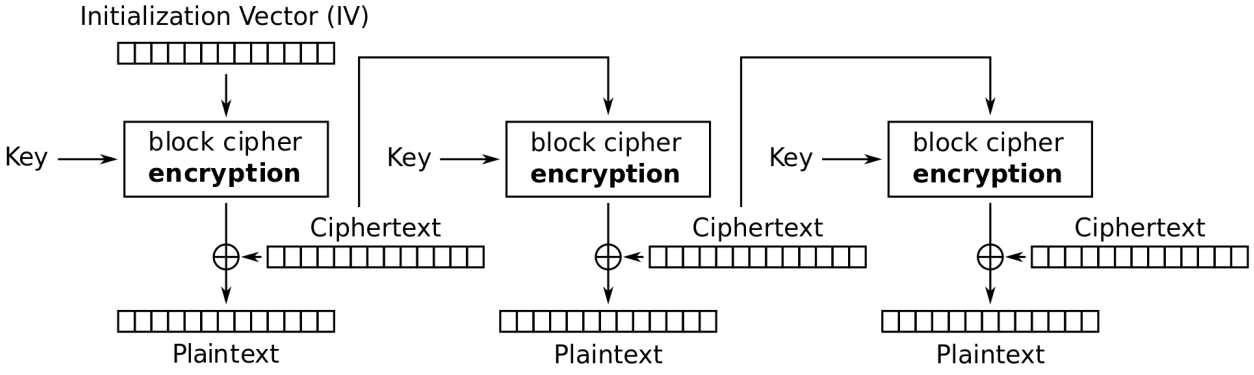

Figure 6: Diagram of decryption for the CFB mode of operation

$$\begin{cases} Z_0 = IV \\ Z_i = E_K(Z_{i-1}) \\ C_i = M_i \oplus Z_i \end{cases}$$

- OFB mode decryption: $P_i = C_i \oplus Z_i$

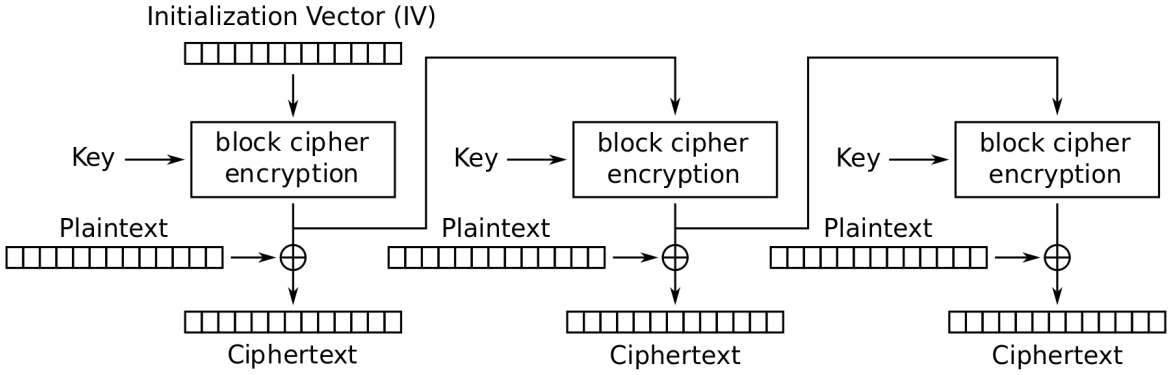

Figure 7: Diagram of encryption for the OFB mode of operation

Counter (CTR) Mode: In CTR mode, a counter is initialized to IV and repeatedly incremented and encrypted to obtain a sequence that can now be used as though they were the keys for a one-time pad: namely, $Z_i = E_K(IV + i)$ and $C_i = Z_i \oplus M_i$. In CTR mode, the IV is sometimes renamed the *nonce*. This is just a terminology difference–nonce and IV can be used interchangeably for the purposes of this class.

Note that in CTR and OFB modes, the decryption algorithm uses the block cipher *encryption* function instead of the decryption function. Intuitively, this is because Alice used the encryption function to generate a one-time pad, so Bob should also use the encryption function to generate the same pad. The plaintext is never passed through the block cipher encryption, so the block cipher decryption is never used.

- CTR mode encryption: $C_i = E_K(IV + i) \oplus M_i$

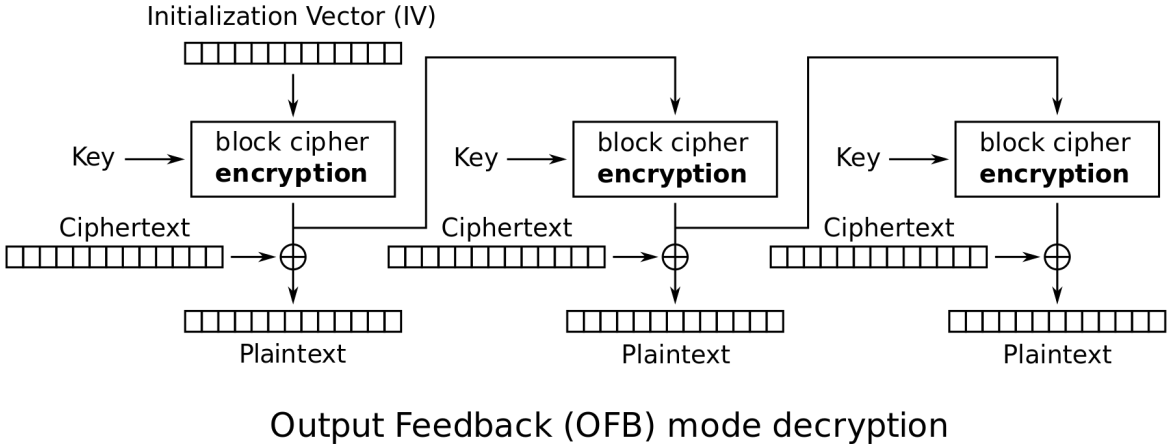

Figure 8: Diagram of decryption for the OFB mode of operation

- CTR mode decryption: $M_i = E_K(IV + i) \oplus C_i$

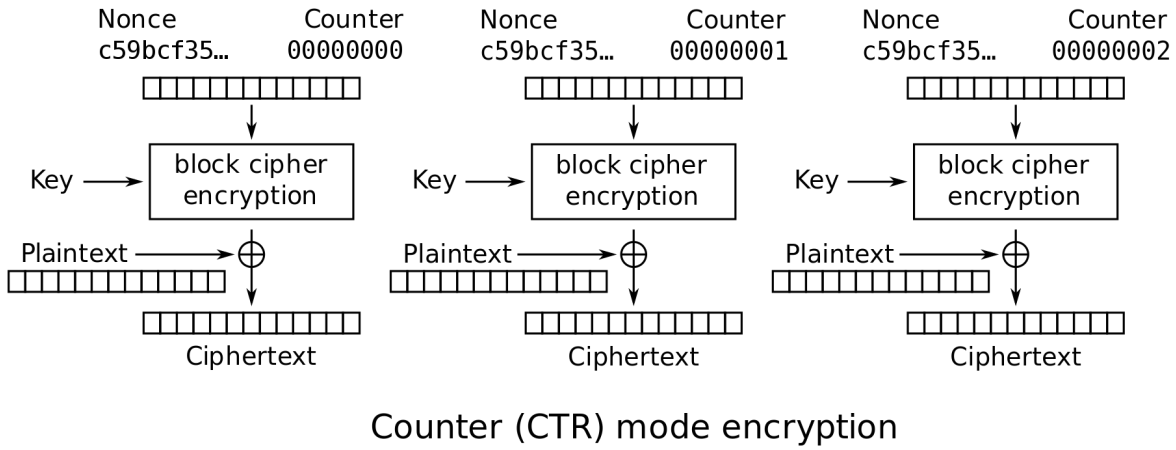

Figure 9: Diagram of encryption for the CTR mode of operation

For the rest of these notes, we will focus on analyzing CBC and CTR modes. As an exercise, you can try performing similar analysis on the other modes as well.

6.7. Parallelization

In some modes, successive blocks must be encrypted or decrypted sequentially. In other words, to encrypt the ith block of plaintext, you first need to encrypt the $i-1$th block of plaintext and see the $i-1$th block of ciphertext output. For high-speed applications, it is often useful to parallelize encryption and decryption.

Of the schemes described above, which ones have parallelizable encryption? Which ones have parallelizable decryption?

CBC mode encryption cannot be parallelized. By examining the encryption equation $C_i = E_K(P_i \oplus C_{i-1})$, we can see that to calculate C_i, we first need to know the value of C_{i-1}. In other words, we have to encrypt the $i-1$th block first before we can encrypt the ith block.

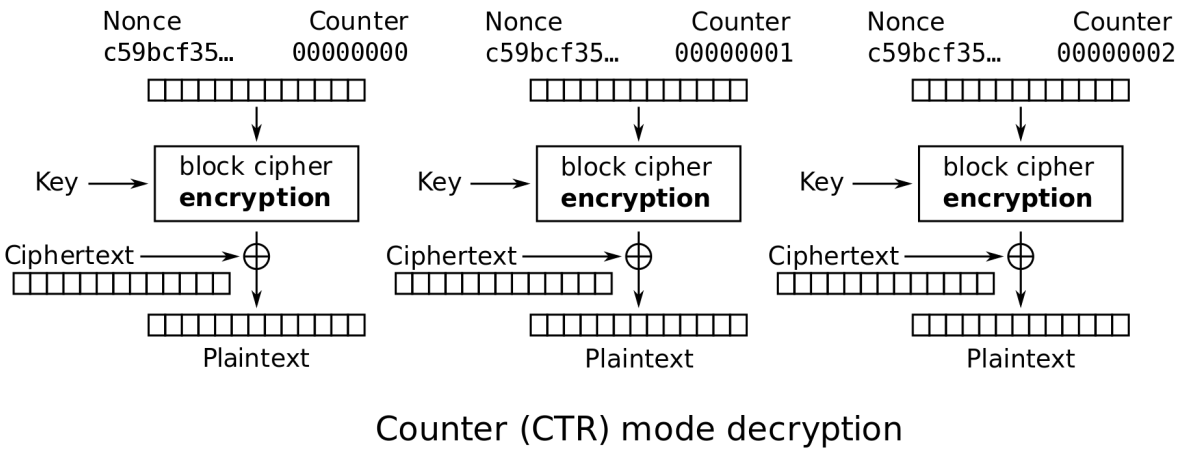

Figure 10: Diagram of decryption for the CTR mode of operation

CBC mode decryption can be parallelized. Again, we examine the decryption equation $P_i = D_K(C_i) \oplus C_{i-1}$. To calculate P_i, we need C_i and C_{i-1}. Neither of these values need to be calculated–when we're decrypting, we already have all of the ciphertext blocks. Thus we can compute all the P_i in parallel.

CTR mode encryption and decryption can both be parallelized. To see this, we can examine the encryption and decryption diagrams. Note that each block cipher only takes the nonce and counter as input, and there is no reliance on any previous ciphertext or plaintext.

6.8. Padding

We have already reasoned that block ciphers let us encrypt messages that are longer than one block long. What happens if we want to send a message that is not a multiple of the block size? It turns out the answer depends on which mode is being used. For this section, assume that the block size is 128 bits, or 16 bytes (16 characters).

In CBC mode, if the plaintext length isn't a multiple of 128 bits, then the last block of plaintext will be slightly shorter than 128 bits. Then the XOR between the 128-bit previous ciphertext and the less-than-128-bit last block of plaintext would be undefined–bitwise XOR only works if the two inputs being XORed are the same length.

Suppose the last block of plaintext is only 100 bits. What if we just XOR the first 100 bits of the previous ciphertext with the 100 bits of plaintext, and ignore the last 28 bits of the previous ciphertext? Now we have a 100-bit input to the block cipher, which only takes 128-bit inputs. This input is undefined for the block cipher.

The solution to this problem is to add padding to the plaintext until it is a multiple of 128 bits.

If we add padding to make the plaintext a multiple of 128 bits, we will need to be able to remove the padding later to correctly recover the original message. Some forms of padding can create ambiguity: for example, consider a padding scheme where we pad a message with all 1s. What happens if we need to pad a message `0000000010111`? We would add 1s until it's a multiple of the block size, e.g. `0000000010111111`. When we try to depad the message, we run into some ambiguity. How many 1s do we remove from the end of the message? It's unclear.

One correct padding scheme is PKCS#7[5] padding. In this scheme, we pad the message by the number of padding bytes used. For example, the message above would be padded as `0000000010111333`, because 3

[5]PKCS stands for Public Key Cryptography Standards.

bytes of padding were needed. To remove the padding, we note that the message ends in a 3, so 3 bytes of padding were used, so we can unambiguously remove the last 3 bytes of padding. Note that if the message is already a multiple of a block size, an entire new block is appended. This way, there is always one unique padding pattern at the end of the message.

Not all modes need padded plaintext input. For example, let's look at CTR mode next. Again, suppose we only have 100 bits in your last block of plaintext. This time, we can actually XOR the 100 bits of plaintext with the first 100 bits of block cipher output, and ignore the last 28 bits of block cipher output. Why? Because the result of the XOR never has to be passed into a block cipher again, so we don't care if it's slightly shorter than 128 bits. The last ciphertext block will just end up being 100 bits instead of 128 bits, and that's okay because it's never used as an input to a block cipher.

How does decryption work? From our encryption step, the last ciphertext block is only 100 bits instead of 128 bits. Then to retrieve the last 100 bits of plaintext, all we have to do is XOR the 100 bits of ciphertext with the first 100 bits of the block cipher output and ignore the last 28 bits of block cipher output.

Recall that CTR mode can be thought of as generating a one-time pad through block ciphers. If the pad is too long, you can just throw away the last few bits of the pad in both the encryption and decryption steps.

6.9. Reusing IVs is insecure

Remember that ECB mode is not IND-CPA secure because it is deterministic. Encrypting the same plaintext twice always results in the same output, and this causes information leakage. All the other modes introduce a random initialization vector (IV) that is different on every encryption in order to ensure that encrypting the same plaintext twice with the same key results in different output.

This also means that when using secure block cipher modes, it is important to always choose a different, random, unpredictable IV for each new encryption. If the same IV is reused, the scheme becomes deterministic, and information is potentially leaked. The severity of information leakage depends on what messages are being encrypted and which mode is being used.

For example, in CTR mode, reusing the IV (nonce) is equivalent to reusing the one-time pad. An attacker who sees two different messages encrypted with the same IV will know the bitwise XOR of the two messages. However, in CBC mode, reusing the IV on two different messages only reveals if two messages start with the same blocks, up until the first difference.

Different modes have different tradeoffs between usability and security. Although proper use of CBC and CTR mode are both IND-CPA, insecure use of either mode (e.g. reusing the IV) breaks IND-CPA security, and the severity of information leakage is different in the two modes. In CBC mode, the information leakage is contained, but in CTR mode, the leakage is catastrophic (equivalent to reusing a one-time pad). On the other hand, CTR mode can be parallelized, but CBC can not, which is why many high performance systems use CTR mode or CTR-mode based encryption schemes.

7. Cryptographic Hashes

7. Cryptographic Hashes

7.1. Overview

A cryptographic hash function is a function, H, that when applied on a message, M, can be used to generate a fixed-length "fingerprint" of the message. As such, any change to the message, no matter how small, will change many of the bits of the hash value with there being no detectable patterns as to how the output changes based on specific input changes. In other words, any changes to the message, M, will change the resulting hash-value in some seemingly random way.

The hash function, H, is deterministic, meaning if you compute $H(M)$ twice with the same input M, you will always get the same output twice. The hash function is unkeyed, as it only takes in a message M and no secret key. This means anybody can compute hashes on any message.

Typically, the output of a hash function is a fixed size: for instance, the SHA256 hash algorithm can be used to hash a message of any size, but always produces a 256-bit hash value.

In a secure hash function, the output of the hash function looks like a random string, chosen differently and independently for each message—except that, of course, a hash function is a deterministic procedure.

To understand the intuition behind hash functions, let's take a look at one of its many uses: document comparisons. Suppose Alice and Bob both have a large, 1 GB document and wanted to know whether the files were the same. While they could go over each word in the document and manually compare the two, hashes provide a quick and easy alternative. Alice and Bob could each compute a hash over the document and securely communicate the hash values to one another. Then, since hash functions are deterministic, if the hashes are the same, then the files must be the same since they have the same "fingerprint". On the other hand, if the hashes are different, it must be the case that the files are different. Determinism in hash functions ensures that providing the same input twice (i.e. providing the same document) will result in the same hash value; however, providing different inputs (i.e. providing two different documents) will result in two different hash values.

7.2. Properties of Hash Functions

Cryptographic hash functions have several useful properties. The most significant include the following:

- **One-way:** The hash function can be computed efficiently: Given x, it is easy to compute $H(x)$. However, given a hash output y, it is infeasible to find any input x such that $H(x) = y$. (This property is also known as "**preimage resistant**.") Intuitively, the one-way property claims that given an output of a hash function, it is infeasible for an adversary to find *any* input that hashes to the given output.
- **Second preimage resistant:** Given an input x, it is infeasible to find another input x' such that $x' \neq x$ but $H(x) = H(x')$. This property is closely related to *preimage resistance*; the difference is that here the adversary also knows a starting point, x, and wishes to tweak it to x' in order to produce the same hash—but cannot. Intuitively, the second preimage resistant property claims that given an input, it is infeasible for an adversary to find another input that has the same hash value as the original input.
- **Collision resistant:** It is infeasible to find *any* pair of messages x, x' such that $x' \neq x$ but $H(x) = H(x')$. Again, this property is closely related to the previous ones. Here, the difference is that the adversary can

freely choose their starting point, x, potentially designing it specially to enable finding the associated x'—but again cannot. Intuitively, the collision resistance property claims that it is infeasible for an adversary to find *any* two inputs that both hash to the same value. While it is impossible to design a hash function that has absolutely no collisions since there are more inputs than outputs (remember the pigeonhole principle), it is possible to design a hash function that makes finding collisions *infeasible* for an attacker.

By "infeasible", we mean that there is no known way to accomplish it with any realistic amount of computing power.

Note, the third property implies the second property. Cryptographers tend to keep them separate because a given hash function's resistance towards the one might differ from its resistance towards the other (where resistance means the amount of computing power needed to achieve a given chance of success).

Under certain threat models, hash functions can be used to verify message integrity. For instance, suppose Alice downloads a copy of the installation disk for the latest version of the Ubuntu distribution, but before she installs it onto her computer, she would like to verify that she has a valid copy of the Ubuntu software and not something that was modified in transit by an attacker. One approach is for the Ubuntu developers to compute the SHA256 hash of the intended contents of the installation disk, and distribute this 256-bit hash value over many channels (e.g., print it in the newspaper, include it on their business cards, etc.). Then Alice could compute the SHA256 hash of the contents of the disk image she has downloaded, and compare it to the hash publicized by Ubuntu developers. If they match, then it would be reasonable for Alice to conclude that she received a good copy of the legitimate Ubuntu software. Because the hash is collision-resistant, an attacker could not change the Ubuntu software while keeping the hash the same. Of course, this procedure only works if Alice has a good reason to believe that she has the correct hash value, and it hasn't been tampered with by an adversary. If we change our threat model to allow the adversary to tamper with the hash, then this approach no longer works. The adversary could simply change the software, hash the changed software, and present the changed hash to Alice.

7.3. Hash Algorithms

Cryptographic hashes have evolved over time. One of the earliest hash functions, MD5 (Message Digest 5) was broken years ago. The slightly more recent SHA1 (Secure Hash Algorithm) was broken in 2017, although by then it was already suspected to be insecure. Systems which rely on MD5 or SHA1 actually resisting attackers are thus considered insecure. Outdated hashes have also proven problematic in non-cryptographic systems. The `git` version control program, for example, identifies identical files by checking if the files produce the same SHA1 hash. This worked just fine until someone discovered how to produce SHA1 collisions.

Today, there are two primary "families" of hash algorithms in common use that are believed to be secure: SHA2 and SHA3. Within each family, there are differing output lengths. SHA-256, SHA-384, and SHA-512 are all instances of the SHA2 family with outputs of 256b, 384b, and 512b respectively, while SHA3-256, SHA3-384, and SHA3-512 are the SHA3 family members.

For the purposes of the class, we don't care which of SHA2 or SHA3 we use, although they are in practice very different functions. The only significant difference is that SHA2 is vulnerable to a *length extension attack*. Given $H(M)$ and the length of the message, but not M itself, there are circumstances where an attacker can compute $H(M\|M')$ for an arbitrary M' of the attacker's choosing. This is because SHA2's output reflects all of its internal state, while SHA3 includes internal state that is never outputted but only used in the calculation of subsequent hashes. While this does not violate any of the aforementioned properties of hash functions, it is undesirable in some circumstances.

In general, we prefer using a hash function that is related to the length of any associated symmetric key algorithm. By relating the hash function's output length with the symmetric encryption algorithm's key length, we can ensure that it is equally difficult for an attacker to break either scheme. For example, if we are using AES-128, we should use SHA-256 or SHA3-256. Assuming both functions are secure, it takes 2^{128} operations to brute-force a 128 bit key and 2^{128} operations to generate a hash collision on a 256 bit hash function. For longer key lengths, however, we may relax the hash difficulty. For example, with 256b AES,

the NSA uses SHA-384, not SHA-512, because, let's face it, 2^{192} operations is already a hugely impractical amount of computation.

7.4. Lowest-hash scheme

Cryptographic hashes have many practical applications outside of cryptography. Here's one example that illustrates many useful properties of cryptographic hashes.

Suppose you are a journalist, and a hacker contacts you claiming to have stolen 150 million (150 million) records from a website. The hacker is keeping the records for ransom, so they don't want to present all 150 million files to you. However, they still wish to prove to you that they have actually stolen 150 million different records, and they didn't steal significantly fewer records and exaggerate the number. How can you be sure that the hacker isn't lying, without seeing all 150 million records?

Remember that the outputs of cryptographic hashes look effectively random–two different inputs, even if they only differ in one bit, give two unpredictably different outputs. Can we use these random-looking outputs to our advantage?

Consider a box with 100 balls, numbered from 1 to 100. You draw a ball at random, observe the value, and put it back. You repeat this n times, then report the lowest number you saw in the n draws. If you drew 10 balls (n=10), you would expect the lowest number to be approximately 10. If you drew 100 balls (n=100), you might expect the lowest number to be somewhere in the range 1-5. If you drew 150 million balls (n=150,000,000), you would be pretty sure that the lowest number was 1. Someone who claims to have drawn 150 million balls and seen a lowest number of 50 has either witnessed an astronomically unlikely event, or is lying about their claim.

We can apply this same idea to hashes. If the hacker hashes all 150 million records, they are effectively generating 150 million unpredictable fixed-length bitstrings, just like drawing balls from the box 150 million times. With some probability calculations (out of scope for this class), we can determine the expected range of the lowest hash values, as well as what values would be astronomically unlikely to be the lowest of 150 million random hashes.

With this idea in mind, we might ask the hacker to hash all 150 million records with a cryptographic hash and return the 10 lowest resulting hashes. We can then check if those hashes are consistent with what we would expect the lowest 10 samples out of 150 million random bitstrings to be. If the hacker only hashed 15 million records and returned the lowest 10 hashes, we should notice that the probability of getting those 10 lowest hashes from 150 million records is astronomically low and conclude that the hacker is lying about their claim.

What if the hacker tries to cheat? If the hacker only has 15 million records, they might try to generate 150 million fake records, hash the fake records, and return the lowest 10 hashes to us. We can make this attack much harder for the attacker by requiring that the attacker also send the 10 records corresponding to the lowest hashes. The hacker won't know which of these 150 million fake records results in the lowest hash, so to guarantee that they can fool the reporter, all 150 million fake records would need to look valid to the reporter. Depending on the setting, this can be very hard or impossible: for example, if we expect the records to be in a consistent format, e.g. `lastname, firstname`, then the attacker would need to generate 150 million fake records that follow the same format.

Still, the hacker might decide to spend some time *precomputing* fake records with low hashes before making a claim. This is called an *offline attack*, since the attacker is generating records offline before interacting with the reporter. We will see more offline attacks when we discuss password hashing later in the notes. We can prevent the offline attack by having the reporter choose a random word at the start of the interaction, like "fubar," and send it to the hacker. Now, instead of hashing each record, the hacker will hash each record, concatenated with the random word. The reporter will give the attacker just enough time to compute 150 million hashes (but no more) before requesting the 10 lowest values. Now, a cheating hacker cannot compute values ahead of time, because they won't know what the random word is.

A slight variation on this method is to hash each record 10 separate times, each with a different reporter-chosen random word concatenated to the end (e.g. "fubar-1," "fubar-2," "fubar-3," etc.). In total, the hacker is now

hashing 1.5b (150 million times 10) records. Then, instead of returning the lowest 10 hashes overall, the hacker returns the record with the lowest hash for each random word. Another way to think of this variation is: the hacker hashes all 150 million records with the first random word concatenated to each record, and returns the record with the lowest hash. Then the hacker hashes all 150 million records again with the second random word concatenated to each record, and returns the record with the lowest hash. This process repeats 10 times until the hacker has presented 10 hashes. The math for using the hash values to estimate the total number of lines is slightly different in this variation (the original uses random selection without substitution, while the variant uses random selection with substitution), but the underlying idea is the same.

8. Message Authentication Codes (MACs)

8. Message Authentication Codes (MACs)

8.1. Integrity and Authenticity

When building cryptographic schemes that guarantee integrity and authentication, the threat we're concerned about is adversaries who send messages pretending to be from a legitimate participant (*spoofing*) or who modify the contents of a message sent by a legitimate participant (*tampering*). To address these threats, we will introduce cryptographic schemes that enable the recipient to detect spoofing and tampering.

In this section, we will define *message authentication codes (MACs)* and show how they guarantee integrity and authenticity. Because MACs are a symmetric-key cryptographic primitive, in this section we can assume that Alice and Bob share a secret key that is not known to anyone else. Later we will see how Alice and Bob might securely exchange a shared secret key over an insecure communication channel, but for now you can assume that only Alice and Bob know the value of the secret key.

8.2. MAC: Definition

A MAC is a keyed checksum of the message that is sent along with the message. It takes in a fixed-length secret key and an arbitrary-length message, and outputs a fixed-length checksum. A secure MAC has the property that any change to the message will render the checksum invalid.

Formally, the MAC on a message M is a value $F(K, M)$ computed from K and M; the value $F(K, M)$ is called the *tag* for M or the MAC of M. Typically, we might use a 128-bit key K and 128-bit tags.

When Alice wants to send a message with integrity and authentication, she first computes a MAC on the message $T = F(K, M)$. She sends the message and the MAC $\langle M, T \rangle$ to Bob. When Bob receives $\langle M, T \rangle$, Bob will recompute $F(K, M)$ using the M he received and check that it matches the MAC T he received. If it matches, Bob will accept the message M as valid, authentic, and untampered; if $F(K, M) \neq T$, Bob will ignore the message M and presume that some tampering or message corruption has occurred.

Note that MACs must be deterministic for correctness–when Alice calculates $T = F(K, M)$ and sends $\langle M, T \rangle$ to Alice, Bob should get the same result when he calculates $F(K, M)$ with the same K and M.

MACs can be used for more than just communication security. For instance, suppose we want to store files on a removable USB flash drive, which we occasionally share with our friends. To protect against tampering with the files on our flash drive, our machine could generate a secret key and store a MAC of each file somewhere on the flash drive. When our machine reads the file, it could check that the MAC is valid before using the file contents. In a sense, this is a case where we are "communicating" to a "future version of ourselves," so security for stored data can be viewed as a variant of communication security.

8.3. MAC: Security properties

Given a secure MAC algorithm F, if the attacker replaces M by some other message M', then the tag will almost certainly[1] no longer be valid: in particular, $F(K, M) \neq F(K, M')$ for any $M' \neq M$.

[1] Strictly speaking, there is a very small chance that the tag for M will also be a valid tag for M'. However, if we choose tags to be long enough—say, 128 bits—and if the MAC algorithm is secure, the chances of this happening should be about $1/2^{128}$, which is small enough that it can be safely ignored.

More generally, there will be no way for the adversary to modify the message and then make a corresponding modification to the tag to trick Bob into accepting the modified message: given M and $T = F(K, M)$, an attacker who does not know the key K should be unable to find a different message M' and a tag T' such that T' is a valid tag on M' (i.e., such that $T' = F(K, M')$). Secure MACs are designed to ensure that even small changes to the message make unpredictable changes to the tag, so that the adversary cannot guess the correct tag for their malicious message M'.

Recall that MACs are deterministic–if Alice calculates $F(K, M)$ twice on the same message M, she will get the same MAC twice. This means that an attacker who sees a pair $M, F(K, M)$ will know a valid MAC for the message M. However, if the MAC is secure, the attacker should be unable to create valid MACs for messages that they have never seen before.

More generally, secure MACs are designed to be secure against known-plaintext attacks. For instance, suppose an adversary Eve eavesdrops on Alice's communications and observes a number of messages and their corresponding tags: $\langle M_1, T_1\rangle, \langle M_2, T_2\rangle, \ldots, \langle M_n, T_n\rangle$, where $T_i = F(K, M_i)$. Then Eve has no hope of finding some new message M' (such that $M' \notin \{M_1, \ldots, M_n\}$) and a corresponding value T' such that T' is the correct tag on M' (i.e., such that $T' = F(K, M')$). The same is true even if Eve was able to choose the M_i's. In other words, even though Eve may know some valid MACs $\langle M_n, T_n\rangle$, she still cannot generate valid MACs for messages she has never seen before.

Here is a formal security definition that captures both properties described above. We imagine a game played between Georgia (the adversary) and Reginald (the referee). Initially, Reginald picks a random key K, which will be used for all subsequent rounds of the game. In each round of the game, Georgia may query Reginald with one of two kinds of queries:

- **Generation query:** Georgia may specify a message M_i and ask for the tag for M_i. Reginald will respond with $T_i = F(K, M_i)$.
- **Verification query:** Alternatively, Georgia may specify a pair of values $\langle M_i, T_i\rangle$ and ask Reginald whether T_i is a valid tag on M_i. Reginald checks whether $T_i \stackrel{?}{=} F(K, M_i)$ and responds "Yes" or "No" accordingly.

Georgia is allowed to repeatedly interact with Reginald in this way. Georgia wins if she ever asks Reginald a verification query $\langle M_n, T_n\rangle$ where Reginald responds "Yes", and where M_n did not appear in any previous generation query to Reginald. In this case, we say that Georgia has successfully forged a tag. If Georgia can successfully forge, then the MAC algorithm is insecure. Otherwise, if there is no strategy that allows Georgia to forge (given a generous allotment of computation time and any reasonable number of rounds of the game), then we say that the MAC algorithm is secure.

This game captures the idea that Georgia the Forger can try to observe the MAC tag on a bunch of messages, but this won't help her forge a valid tag on any new message. In fact, even if Georgia carefully selects a bunch of chosen messages and gets Alice to transmit those messages (i.e., she gets Alice to compute the MAC on those messages with her key, and then transmit those MAC tags), it still won't help Georgia forge a valid tag on any new message. Thus, MACs provide security against chosen-plaintext/ciphertext attacks, the strongest threat model.

8.4. AES-EMAC

How do we build secure MACs?

There are a number of schemes out there, but one good one is AES-CMAC, an algorithm standardized by NIST. Instead of showing you AES-CMAC, we'll look at a related algorithm called AES-EMAC. AES-EMAC is a slightly simplified version of AES-CMAC that retains its essential character but differs in a few details.

In AES-EMAC, the key K is 256 bits, viewed as a pair of 128-bit AES keys: $K = \langle K_1, K_2\rangle$. The message M is decomposed into a sequence of 128-bit blocks: $M = P_1 \| P_2 \| \ldots \| P_n$. We set $S_0 = 0$ and compute

$$S_i = \mathrm{AES}_{K_1}(S_{i-1} \oplus P_i), \qquad \text{for } i = 1, 2, \ldots, n.$$

Finally we compute $T = \text{AES}_{K_2}(S_n)$; T is the tag for message M. Here is what it looks like:

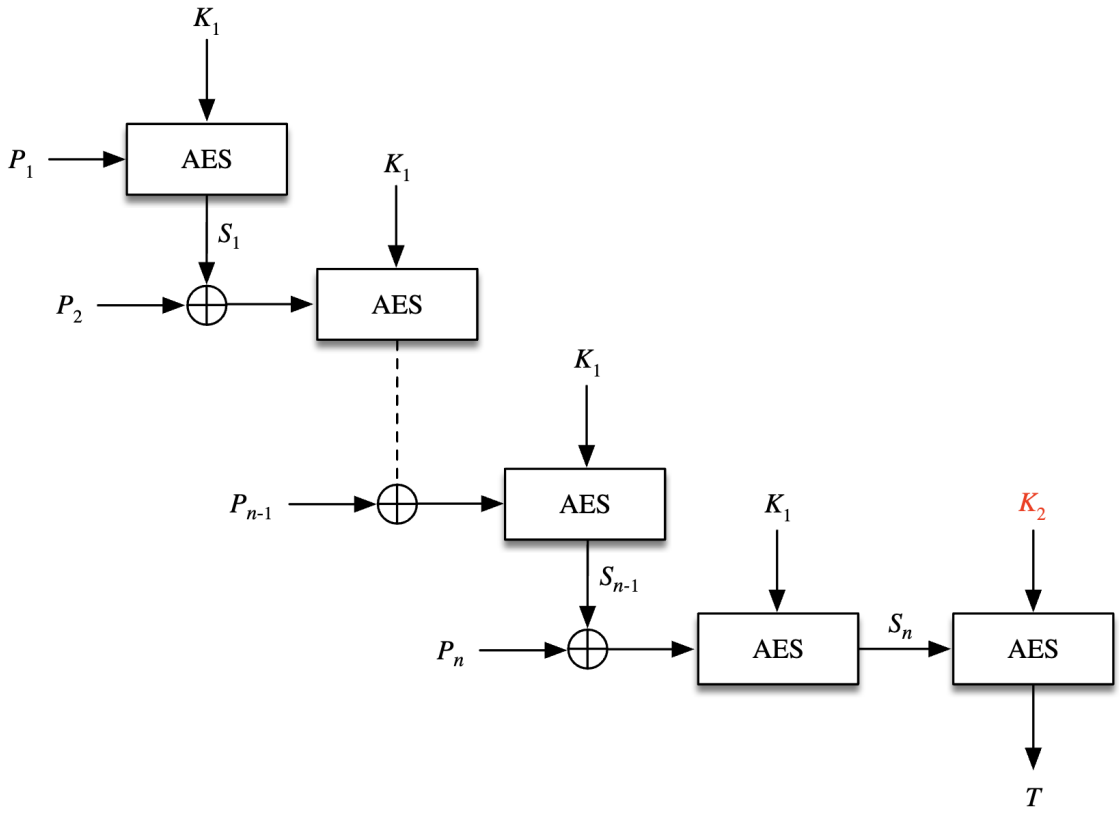

Figure 1: AES-EMAC block diagram, with K2 highlighted as the final encryption step

Assuming AES is a secure block cipher, this scheme is provably secure, using the unforgeability definition and security game described in the previous section. An attacker cannot forge a valid AES-EMAC for a message they haven't seen before, even if they are allowed to query for MACs of other messages.

8.5. HMAC

One of the best MAC constructions available is the HMAC, or Hash Message Authentication Code, which uses the cryptographic properties of a cryptographic hash function to construct a secure MAC algorithm.

HMAC is an excellent construction because it combines the benefits of both a MAC and the underlying hash. Without the key, the tag does not leak information about the message. Even with the key, it is computationally intractable to reconstruct the message from the hash output.

There are several specific implementations of HMAC that use different cryptographic hash functions: for example, HMAC_SHA256 uses SHA256 as the underlying hash, while HMAC_SHA3_256 uses SHA3 in 256 bit mode as the underlying hash function. The choice of underlying hash depends on the application. For example, if we are using HMACs with a block cipher, we would want to choose an HMAC whose output is twice the length of the keys used for the associated block cipher, so if we are encrypting using AES_192 we should use HMAC_SHA_384 or HMAC_SHA3_384.

The output of HMAC is the same number of bits as the underlying hash function, so in both of these implementations it would be 256 bits of output. In this section, we'll denote the number of bits in the hash output as n.

To construct the HMAC algorithm, we first start with a more general version, NMAC:

$\text{NMAC}(K_1, K_2, M) = H(K_1 \| H(K_2 \| M))$

In words, NMAC concatenates K_2 and M, hashes the result, concatenates the result with K_1, and then hashes that result.

Note that NMAC takes two keys, K_1 and K_2, both of length n (the length of the hash output). If the underlying hash function H is cryptographic and K_1 and K_2 are unrelated[2], then NMAC is provably secure.

HMAC is a more specific version of NMAC that only requires one key instead of two unrelated keys:

$\text{HMAC}(M, K) = H((K' \oplus opad) \| H((K' \oplus ipad) \| M))$

The HMAC algorithm actually supports a variable-length key K. However, NMAC uses K_1 and K_2 that are the same length as the hash output n, so we first transform K to be length n. If K is shorter than n bits, we can pad K with zeros until it is n bits. If K is longer than n bits, we can hash K to make it n bits. The transformed n-bit version of K is now denoted as K'.

Next, we derive two unrelated keys from K'. It turns out that XORing K' with two different pads is sufficient to satisfy the definition of "unrelated" used in the NMAC security proof. The HMAC algorithm uses two hardcoded pads, *opad* (outer pad) and *ipad* (inner pad), to generate two unrelated keys from a single key. The first key is $K_1 = K' \oplus opad$, and the second key is $K_2 = K' \oplus ipad$. *opad* is the byte `0x5c` repeated until it reaches n bits. Similarly, *ipad* is the byte `0x36` repeated until it reaches n bits.[3]

In words, HMAC takes the key and pads it or hashes it to length n. Then, HMAC takes the resulting modified key, XORs it with the ipad, concatenates the message, and hashes the resulting string. Next, HMAC takes the modified key, XORs it with the opad, and then concatenates it to the previous hash. Hash this final string to get the result.

Because NMAC is provably secure, and HMAC is a special case of NMAC that generates the two unrelated keys from one key, HMAC is also provably secure. This proof assumes that the underlying hash is a secure cryptographic hash, which means if you can find a way to break HMAC (forge a valid HMAC without knowing the key), then you have also broken the underlying cryptographic hash.

Because of the properties of a cryptographic hash, if you change just a single bit in either the message or the key, the output will be a completely different, unpredictable value. Someone who doesn't know the key won't be able to generate tags for arbitrary messages. In fact, they can't even distinguish the tag for a message from a random value of the same length.

HMAC is also very efficient. The inner hash function call only needs to hash the bits of the message, plus n bits, and the outer hash function call only needs to hash $2n$ bits.

8.6. MACs are not confidential

A MAC does not guarantee confidentiality on the message M to which it is applied. In the examples above, Alice and Bob have been exchanging non-encrypted plaintext messages with MACs attached to each message. The MACs provide integrity and authenticity, but they do nothing to hide the contents of the actual message. In general, MACs have no confidentiality guarantees–given $F(K, M)$, there is no guarantee that the attacker cannot learn something about M.

As an example, we can construct a valid MAC that guarantees integrity but does not guarantee confidentiality. Consider the MAC function F' defined as $F'(K, M) = F(K, M) \| M$. In words, F' contains a valid MAC of the message, concatenated with the message plaintext. Assuming F is a valid MAC, then F' is also valid MAC. An attacker who doesn't know K won't be able to generate $F'(K, M')$ for the attacker's message M', because they won't be able to generate $F(K, M')$, which is part of $F'(K, M')$. However, F' does not provide any confidentiality on the message–in fact, it leaks the entire message!

[2]The formal definition of "unrelated" is out of scope for these notes. See this paper to learn more.

[3]The security proof for HMAC just required that ipad and opad be different by at least one bit but, showing the paranoia of cryptography engineers, the designers of HMAC chose to make them very different.

There is no notion of "reversing" or "decrypting" a MAC, because both Alice and Bob use the same algorithm to generate MACs. However, there is nothing that says a MAC algorithm can't be reversed if you know the key. For example, with AES-MAC it is clear that if the message is a single block, you can run the algorithm in reverse to go from the tag to the message. Depending on the particular MAC algorithm, this notion of reversing a MAC might also lead to leakage of the original message.

There are some MAC algorithms that don't leak information about the message because of the nature of the underlying implementation. For example, if the algorithm directly applies a block cipher, the block cipher has the property that it does not leak information about the plaintext. Similarly, HMAC does not leak information about the message, since it maintains the properties of the cryptographic hash function.

In practice, we usually want to guarantee confidentiality in addition to integrity and authenticity. Next we will see how we can combine encryption schemes with MACs to achieve this.

8.7. Authenticated Encryption

An *authenticated encryption* scheme is a scheme that simultaneously guarantees confidentiality and integrity on a message. As you might expect, symmetric-key authenticated encryption modes usually combine a block cipher mode (to guarantee confidentiality) and a MAC (to guarantee integrity and authenticity).

Suppose we have an IND-CPA secure encryption scheme Enc that guarantees confidentiality, and an unforgeable MAC scheme MAC that guarantees integrity and authenticity. There are two main approaches to authenticated encryption: encrypt-then-MAC and MAC-then-encrypt.

In the *encrypt-then-MAC* approach, we first encrypt the plaintext, and then produce a MAC over the ciphertext. In other words, we send the two values $\langle \mathsf{Enc}_{K_1}(M), \mathsf{MAC}_{K_2}(\mathsf{Enc}_{K_1}(M)) \rangle$. This approach guarantees *ciphertext integrity*–an attacker who tampers with the ciphertext will be detected by the MAC on the ciphertext. This means that we can detect that the attacker has tampered with the message without decrypting the modified ciphertext. Additionally, the original message is kept confidential since neither value leaks information about the plaintext. The MAC value might leak information about the ciphertext, but that's fine; we already know that the ciphertext doesn't leak anything about the plaintext.

In the *MAC-then-encrypt* approach, we first MAC the message, and then encrypt the message and the MAC together. In other words, we send the value $\mathsf{Enc}_{K_1}(M \| \mathsf{MAC}_{K_2}(M))$. Although both the message and the MAC are kept confidential, this approach does not have ciphertext integrity, since only the original message was tagged. This means that we'll only detect if the message is tampered after we decrypt it. This may not be desirable in some applications, because you would be running the decryption algorithm on arbitrary attacker inputs.

Although both approaches are theoretically secure if applied correctly, in practice, the MAC-then-Encrypt approach has been attacked through side channel vectors. In a side channel attack, improper usage of a cryptographic scheme causes some information to leak through some other means besides the algorithm itself, such as the amount of computation time taken or the error messages returned. One example of this attack was a padding oracle attack against a particular TLS implementation using the MAC-then-encrypt approach. Because of the possibility of such attacks, encrypt-then-MAC is generally the better approach.

In both approaches, the encryption and MAC functions should use different keys, because using the same key in an authenticated encryption scheme makes the scheme vulnerable to a large category of potential attacks. These attacks take advantage of the fact that two different algorithms are called with the same key, as well as the properties of the particular encryption and MAC algorithms, to potentially leak information about the original message. The easiest way to avoid this category of attacks is to simply use different keys for the encryption and MAC functions.

8.8. AEAD Encryption Modes

There are also some special block cipher operation modes, known as AEAD (Authenticated Encryption with Additional Data) that, in addition to providing confidentiality like other appropriate block cipher modes, also provide integrity/authenticity.

The “additional data” component means that the integrity is provided not just over the encrypted portions of the message but some additional unencrypted data. For example, if Alice wants to send a message to Bob, she may want to include that the message is "From Alice to Bob" in plaintext (for the benefit of the system that routes the message from Alice to Bob) but also include it in the set of data protected by the authentication.

While powerful, using these modes improperly will lead to catastrophic failure in security, since a mistake will lead to a loss of both confidentiality and integrity at the same time.

One such mode is called AES-GCM (Galois Counter Mode). The specifics are out of scope for these notes, but at a high level, AES-GCM is a stream cipher that operates similarly to AES-CTR (counter) mode. The security properties of AES-GCM are also similar to CTR–in particular, IV reuse also destroys the security of AES-GCM. Since the built-in MAC in AES-GCM is also a function of the CTR mode encryption, improper use of AES-GCM causes loss of both confidentiality and integrity.

Some other modes include CCM mode, CWC mode, and OCB mode, but these are out of scope for these notes.

9. Pseudorandom Number Generators

9. Pseudorandom Number Generators

9.1. Randomness and entropy

As we've seen in the previous sections, cryptography often requires randomness. For example, symmetric keys are usually randomly-generated bits, and random IVs and nonces are required to build secure block cipher chaining modes.

In cryptography, when we say "random," we usually mean "random and unpredictable." For example, flipping a biased coin that comes up heads 99% of the time is random, but you can predict a pattern–for a given coin toss, if you guess heads, it's very likely you're correct. A better source of randomness for cryptographic purposes would be flipping a fair coin, because the outcome is harder to predict than the outcome of the biased coin flip. Consider generating a random symmetric key: you would want to use outcomes of the fair coin to generate the key, because that makes it harder for the attacker to guess your key than if you had used outcomes of the biased coin to generate the key.

We can formalize this concept of unpredictability by defining *entropy*, a measure of uncertainty or surprise, for any random event. The biased coin has low entropy because you expect a given outcome most of the time. The fair coin has high entropy because you are very uncertain about the outcome. The specifics of entropy are beyond the scope of this class, but an important note is that the uniform distribution (all outcomes equally likely) produces the greatest entropy. In cryptography, we generally want randomness with the most entropy, so ideally, any randomness should be bits drawn from a uniform distribution (i.e. the outcomes of fair coin tosses).

However, true, unbiased randomness is computationally expensive to generate. True randomness usually requires sampling data from an unpredictable physical process, such as an unpredictable circuit on a CPU, random noise signals, or the microsecond at which a user presses a key. These sources may be biased and predictable, making it even more challenging to generate unbiased randomness.

Instead of using expensive true randomness each time a cryptographic algorithm requires randomness, we instead use *pseudo-randomness*. Pseudorandom numbers are generated deterministically using an algorithm, but they look random. In particular, a good pseudorandom number algorithm generates bits that are *computationally indistinguishable* from true random bits–there is no efficient algorithm that would let an attacker distinguish between pseudorandom bits and truly random bits.

9.2. Pseudorandom Number Generators (pRNGs)

A *pseudorandom number generator (pRNG)* is an algorithm that takes a small amount of truly random bits as input and outputs a long sequence of pseudorandom bits. The initial truly random input is called the *seed*.

The pRNG algorithm is deterministic, so anyone who runs the pRNG with the same seed will see the same pseudorandom output. However, to an attacker who doesn't know the seed, the output of a secure pRNG is computationally indistinguishable from true random bits. A pRNG is not completely indistinguishable from true random bits–given infinite computational time and power, an attacker can distinguish pRNG output from truly random output. If the pRNG takes in an n-bit seed as input, the attacker just has to input all 2^n possible seeds and see if any of the 2^n outputs matches the bitstring they received. However, when restricted

to any practical computation limit, an attacker has no way of distinguishing pRNG output from truly random output.

It would be very inefficient if a pRNG only outputted a fixed number of pseudorandom bits for each truly random input. If this were the case, we would have to generate more true randomness each time the pRNG output has all been used. Ideally, we would like the pRNG to take in an initial seed and then be available to generate as many pseudorandom bits as needed on demand. To achieve this, the pRNG maintains some internal state and updates the state any time the pRNG generates new bits or receives a seed as input.

Formally, a pRNG is defined by the following three functions:

- *Seed(entropy)*: Take in some initial truly random entropy and initialize the pRNG's internal state.
- *Reseed(entropy)*: Take in some additional truly random entropy, updating the pRNG's internal state as needed.
- *Generate(n)*: Generate n pseudorandom bits, updating the internal state as needed. Some pRNGs also support adding additional entropy directly during this step.

9.3. Rollback resistance

In the previous section, we defined a secure pRNG as an algorithm whose output is computationally indistinguishable from random if the attacker does not know the seed and internal state. However, this definition does not say anything about the consequences of an attacker who does manage to compromise the internal state of a secure pRNG.

An additional desirable property of a secure pRNG is *rollback resistance*. Suppose a pRNG has been used to generate 100 bits, and an attacker is somehow able to learn the internal state immediately after bit 100 has been generated. If the pRNG is rollback-resistant, then the attacker cannot deduce anything about any previously-generated bit. Formally, the previously-generated output of the pRNG should still be computationally indistinguishable from random, even if the attacker knows the current internal state of the pRNG.

Not all secure pRNGs are rollback-resistant, but rollback resistance is an important property for any practical cryptographic pRNG implementation. Consider a cryptosystem that uses a single pRNG to generate both the secret keys and the IVs (or nonces) for a symmetric encryption scheme. The pRNG is first used to generate the secret keys, and then used again to generate the IVs. If this pRNG was not rollback-resistant, then an attacker who compromises the internal state at this point could learn the value of the secret key.

9.4. HMAC-DRBG

There are many implementations of pRNGs, but one commonly-used pRNG in practice is HMAC-DRBG[1], which uses the security properties of HMAC to build a pRNG.

HMAC-DRBG maintains two values as part of its internal state, K and V. K is used as the secret key to the HMAC, and V is used as the "message" input to the HMAC.

To generate a block of pseudorandom bits, HMAC-DRBG computes HMAC on the previous block of pRNG output. This can be repeated to generate as many pseudorandom bits as needed. Recall that the output of HMAC looks random to an attacker who doesn't know the key. As long as we keep the internal state (which includes K) secret, an attacker cannot distinguish the output of the HMAC from random bits, so the pRNG is secure.

We also use HMAC to update the internal state K and V each time. If additional true randomness is provided, we add it to the "message" input to HMAC.

[1] DRBG stands for Deterministic Random Bit Generator

Algorithm 1 Generate(n): Generate n pseudorandom bits, with no additional true random input

output = ‘ ’

while len(output) $< n$ **do**

$V = \text{HMAC}(K, V)$

output = output$\|V$

end while

$K = \text{HMAC}(K, V\|\texttt{0x00})$

$V = \text{HMAC}(K, V)$

return output$[0 : n]$

At line 3, we are repeatedly calling HMAC on the previous block of output. The while loop repeats this process until we have at least n bits of output. Once we have enough output, we update the internal state with two additional HMAC calls, and then return the first n bits of pseudorandom output.

Next, let's see how to seed the pRNG. The seed and reseed algorithms use true randomness as input to the HMAC, and uses the output of slight variations on the HMAC input to update K and V.

Algorithm 2 Seed(s): Take some truly random bits s and initialize the internal state.

$K = 0$

$V = 0$

$H = \text{HMAC}(K, V\|s\|\texttt{0x00})$

$V = \text{HMAC}(K, V)$

$K = \text{HMAC}(K, V\|s\|\texttt{0x01})$

$V = \text{HMAC}(K, V)$

The reseed algorithm is identical to the seed algorithm, except we don't need to reset K and V to 0 (steps 1-2).

Finally, if we want to generate pseudorandom output and add entropy at the same time, we combine the two algorithms above:

Algorithm 3 Generate(n): Generate n pseudorandom bits, with additional true random input s.

output = ‘ ’

while len(output) $< n$ **do**

$V = \text{HMAC}(K, V)$

output = output$\|V$

end while

$K = \text{HMAC}(K, V\|s\|\texttt{0x00})$

$V = \text{HMAC}(K, V)$

$K = \text{HMAC}(K, V\|s\|\texttt{0x01})$

$V = \text{HMAC}(K, V)$

return output$[0 : n]$

The specific design decisions of HMAC-DRBG, such as why it uses `0x00` and `0x01`, are not so important. The main takeaway is that because HMAC output is indistinguishable from random, the output of HMAC-DRBG (which is essentially lots of HMAC outputs) is also indistinguishable from random.

The use of the cryptographic hash function in both the seeding and reseeding algorithms means that HMAC-DRBG can accept an arbitrary long initial seed. For example, if each bit of the input seed really only has 0.1 bits of entropy (e.g. because it is a highly biased coin), using 2560 bits of seed material will leave HMAC-DRBG with 256b of actual entropy for its internal operations. Furthermore, adding in additional strings that contain *no* entropy (such as a string of 0 bits or the number π) doesn't make the internal state worse.

Additionally, HMAC-DRBG has rollback resistance: if you can compute the previous state from the current state you have successfully reversed the underlying hash function!

9.5. Stream ciphers

As we've seen in the previous section, an attacker without knowledge of the internal state of a secure, rollback-resistant pRNG cannot predict the pRNG's past or future output, and the attacker cannot distinguish the pRNG output from random bits. This sounds very similar to the properties we want in a random, unpredictable one-time pad. In fact, we can use pRNGs to generate a one-time pad that we then use for encrypting messages. This encryption scheme is an example of a class of algorithms known as *stream ciphers*.

Recall that in block ciphers, we encrypted and decrypted messages by splitting them into fixed-size blocks. Stream ciphers use a different approach to encryption, in which we encrypt and decrypt messages as they arrive, one bit at a time. You can imagine a stream cipher operating on an encrypted file being downloaded from the Internet: as each subsequent bit is downloaded, the stream cipher can immediately decrypt the bit while waiting for the next bit to download. This is different from a block cipher, where you might need a block of bits, several blocks of bits, or the entire message to be downloaded before you can start decrypting.

A common class of stream cipher algorithms involves outputting an unpredictable stream of bits, and then using this stream as the key of a one-time pad. In other words, each bit of the plaintext message is XORed with the corresponding bit in the key stream.

The output of a secure pRNG can be used as the key for this one-time pad scheme. Formally, in a pRNG-based stream cipher, the secret key is the initial seed used to seed the pRNG. To encrypt an n-bit message, Alice runs the pRNG until it generates n pseudorandom bits. Then she XORs the pseudorandom bits with the plaintext message. Since the pRNG can generate as many bits as needed, this algorithm can encrypt arbitrary-length messages.

To decrypt the message, Bob uses the same secret key to seed the pRNG. Since the pRNG is deterministic with the same seed, Bob will generate the same pseudorandom bits when he runs the pRNG. Then Bob XORs the pseudoranom bits with the ciphertext to learn the original plaintext.

To avoid key reuse, Alice and Bob can both seed the pRNG with a random IV in addition to their secret key so that the pRNG output is different and unpredictable each time. In short, the pRNG algorithm is:

- Encryption: $Enc(K, M) = \langle IV, PRNG(K, IV) \oplus M \rangle$
- Decryption: $Dec(K, IV, C_2) = PRNG(K, IV) \oplus C_2$

AES-CTR is effectively a stream cipher. Although technically AES appears to be a pseudo-random permutation rather than a pseudo-random generator, in practice the results are similar. As long as the total ciphertext encrypted with a given key is kept to a reasonable level (2^{64}b), the one-time pad output of AES-CTR should be effectively indistinguishable from pRNG output. Beyond this threshold, there is a significant probability

with CTR mode that there will be two blocks with identical ciphertext, which would leak information that the underlying plaintext blocks are different.

Although theoretically we could use any cryptographically secure pRNG (like HMAC-DRBG) as a stream cipher, dedicated stream ciphers (such as the ChaCha20 cipher) have properties that we would consider a disadvantage in a secure pRNG but are actually advantages for stream ciphers. In particular, both AES-CTR mode encryption and ChaCha20 include a counter value in the computation of the stream.

One desirable consequence of including a counter is the ability to encrypt or decrypt an arbitrary point in the message without starting from the beginning. If you have a 1 terabyte file encrypted using either AES-CTR mode or ChaCha20 and you wish to read just the last bytes, you can set the counter to the appropriate point and just decrypt the last bytes, while if you used HMAC-DRBG as the stream cipher, you would need to start at the beginning of the message and compute 1 terabytes of HMAC-DRBG output before you could read the end of the file.[2]

[2]This use of a counter value means that if you view it as a pRNG, AES-CTR and ChaCha20 lack rollback resistance as the key and counter are the internal state. But on the other hand, it is this ability to rollback and jump-forward into the output space that makes them more useful as stream ciphers.

10. Diffie-Hellman Key Exchange

10. Diffie-Hellman key exchange

In the previous sections, we discussed symmetric-key schemes such as block ciphers and MACs. For these schemes to work, we assumed that Alice and Bob both share a secret key that no one else knows. But how would they be able to exchange a secret key if they can only communicate through an insecure channel? It turns out there is a clever way to do it, first discovered by Whit Diffie and Martin Hellman in the 1970s.

The goal of Diffie-Hellman is usually to create an *ephemeral* key. An ephemeral key is used for some series of encryptions and decryptions and is discarded once it is no longer needed. Thus Diffie-Hellman is effectively a way for two parties to agree on a *random* value in the face of an eavesdropper.

10.1. Diffie-Hellman intuition

A useful analogy to gain some intuition about Diffie-Hellman key exchange is to think about colors. Alice and Bob want to both share a secret color of paint, but Eve can see any paint being exchanged between Alice and Bob. It doesn't matter what the secret color is, as long as only Alice and Bob know it.

Alice and Bob each start by deciding on a secret color–for example, Alice generates amber and Bob generates blue. If Eve wasn't present, Alice and Bob could just send their secret colors to each other and mix the two secret colors together to get the final color. However, since Eve can see any colors sent between Alice and Bob, they must somehow hide their colors when exchanging them.

To hide their secret colors, Alice and Bob agree on a publicly-known common paint color–in this example, green. They each take their secret color, mix it with the common paint color, and then send it over the insecure channel. Alice sends a green-amber mixture to Bob, and Bob sends a green-blue mixture to Alice. Here we're assuming that given a paint mixture, Eve cannot separate the mixture into its original colors.

At this point, Alice and Bob have each other's secret colors, mixed with the common color. All that's left is to add their own secret. Alice receives the green-blue mixture from Bob and adds her secret amber to get green-blue-amber. Bob receives the green-amber mixture from Alice and adds his secret blue to get green-amber-blue. Alice and Bob now both have a shared secret color (green-amber-blue).

The only colors exchanged over the insecure channel were the secret mixtures green-amber and green-blue. Given green-amber, Eve would need to add blue to get the secret, but she doesn't know Bob's secret color, because it wasn't sent over the channel, and she can't separate the green-blue mixture. Eve could also try mixing the green-amber and green-blue mixtures, but you can imagine this wouldn't result in exactly the same secret, since there would be too much green in the mix. The result would be more like green-amber-green-blue than green-amber-blue.

10.2. Discrete logarithm problem

The secret exchange in the color analogy relied on the fact that mixing two colors is easy, but separating a mixture of two colors is practically impossible. It turns out that there is a mathematical equivalent of this. We call these *one-way functions*: a function f such that given x, it is easy to compute $f(x)$, but given y, it is practically impossible to find a value x such that $f(x) = y$.

A one-way function is also sometimes described as the computational equivalent of a process that turns a cow into hamburger: given the cow, you can produce hamburger, but there's no way to restore the original cow from the hamburger.

There are many functions believed to be one-way functions. The simplest one is exponentiation modulo a prime: $f(x) = g^x \pmod{p}$, where p is a large prime and g is a specially-chosen generator[1].

Given x, it is easy to calculate $f(x)$ (you may recall the repeated squaring algorithm from CS 70). However, given $f(x) = g^x \pmod{p}$, there is no known efficient algorithm to solve for x. This is known as the *discrete logarithm problem*, and it is believed to be computationally hard to solve.

Using the hardness of the discrete log problem and the analogy from above, we are now ready to construct the Diffie-Hellman key exchange protocol.

10.3. Diffie-Hellman protocol

In high-level terms, the Diffie-Hellman key exchange works like this.

Alice and Bob first establish the public parameters p and g. Remember that p is a large prime and g is a generator in the range $1 < g < p - 1$. For instance, Alice could pick p and g and then announce it publicly to Bob. Today, g and p are often hardcoded or defined in a standard so they don't need to be chosen each time. These values don't need to be specific to Alice or Bob in any way, and they're not secret.

Then, Alice picks a secret value a at random from the set $\{0, 1, \ldots, p-2\}$, and she computes $A = g^a \bmod p$. At the same time, Bob randomly picks a secret value b and computes $B = g^b \bmod p$.

Now Alice announces the value A (keeping a secret), and Bob announces B (keeping b secret). Alice uses her knowledge of B and a to compute

$S = B^a = (g^b)^a = g^{ba} \pmod{p}$.

Symmetrically, Bob uses his knowledge of A and b to compute

$S = A^b = (g^a)^b = g^{ab} \pmod{p}$.

Note that $g^{ba} = g^{ab} \pmod{p}$, so both Alice and Bob end up with the same result, S.

Finally, Alice and Bob can use S as a shared key for a symmetric-key cryptosystem (in practice, we would apply some hash function to S first and use the result as our shared key, for technical reasons).

The amazing thing is that Alice and Bob's conversation is entirely public, and from this public conversation, they both learn this secret value S—yet eavesdroppers who hear their entire conversation cannot learn S.

As far as we know, there is no efficient algorithm to deduce $S = g^{ab} \bmod p$ from the values Eve sees, namely $A = g^a \bmod p$, $B = g^b \bmod p$, g, and p. The hardness of this problem is closely related to the discrete log problem discussed above. In particular, the fastest known algorithms for solving this problem take $2^{cn^{1/3}(\log n)^{2/3}}$ time, if p is a n-bit prime. For $n = 2048$, these algorithms are far too slow to allow reasonable attacks.

Here is how this applies to secure communication among computers. In a computer network, each participant could pick a secret value x, compute $X = g^x \bmod p$, and publish X for all time. Then any pair of participants who want to hold a conversation could look up each other's public value and use the Diffie-Hellman scheme to agree on a secret key known only to those two parties. This means that the work of picking p, g, x, and X can be done in advance, and each time a new pair of parties want to communicate, they each perform only one modular exponentiation. Thus, this can be an efficient way to set up shared keys.

Here is a summary of Diffie-Hellman key exchange:

- **System parameters:** a 2048-bit prime p, a value g in the range $2 \ldots p-2$. Both are arbitrary, fixed, and public.

[1] You don't need to worry about how to choose g, just know that it satisfies some special number theory properties. In short, g must satisfy the following properties: $1 < g < p - 1$, and there exists a k where $g^k = a$ for all $1 \le a \le p - 1$.

- **Key agreement protocol:** Alice randomly picks a in the range $1 \ldots p-2$ and sends $A = g^a \bmod p$ to Bob. Bob randomly picks b in the range $1 \ldots p-2$ and sends $B = g^b \bmod p$ to Alice. Alice computes $K = B^a \bmod p$. Bob computes $K = A^b \bmod p$. Alice and Bob both end up with the same random secret key K, yet as far as we know no eavesdropper can recover K in any reasonable amount of time.

10.4. Elliptic-curve Diffie-Hellman

In the section above, we used the discrete log problem to construct a Diffie-Hellman protocol, but we can generalize Diffie-Hellman key exchange to other one-way functions. One commonly used variant of Diffie-Hellman relies on the elliptic curve discrete logarithm problem, which is based on the math around elliptic curves instead of modular arithmetic. Although the underlying number theory is more complicated, elliptic-curve Diffie-Hellman can use smaller keys than modular arithmetic Diffie-Hellman and still provide the same security, so it has many useful applications.

For this class, you don't need to understand the underlying math that makes elliptic-curve Diffie-Hellman work, but you should know that the idea behind the protocol is the same.

Alice and Bob start with a publicly known point on the elliptic curve G. Alice chooses a secret integer a and Bob chooses a secret integer b.

Alice computes $A = a \cdot G$ (this is a point on the curve A, obtained by adding the point G to itself a times), and Bob computes $B = b \cdot G$. Alice sends A to Bob, and Bob sends B to Alice.

Alice computes

$S = a \cdot B = a \cdot b \cdot G$

and Bob computes

$S = b \cdot A = b \cdot a \cdot G$

Because of the properties of the elliptic curve, Alice and Bob will derive the same point S, so they now have a shared secret. Also, the elliptic-curve Diffie-Hellman problem states that given $A = a \cdot G$ and $B = b \cdot G$, there is no known efficient method for Eve to calculate S.

10.5. Difficulty in Bits

It is generally believed that the discrete log problem is hard, but how hard? In practice, it is generally believed that computing the discrete log modulo a 2048b prime, computing the elliptic curve discrete log on a 256b curve, and brute forcing a 128b symmetric key algorithm are all roughly the same difficulty. (Brute-forcing the 128b key is believed to be slightly harder than the other two.)

Thus, if we are using Diffie-Hellman key exchange with other cryptoschemes, we try to relate the difficulty of the schemes so that it is equally difficult for an attacker to break any scheme. For example, 128b AES tends to be used with SHA-256 and either 256b elliptic curves or 2048b primes for Diffie-Hellman. Similarly, for top-secret use, the NSA uses 256b AES, 384b Elliptic Curves, SHA-384, and 3096b Diffie-Hellman and RSA.

10.6. Attacks on Diffie-Hellman

As we've seen, Diffie-Hellman is secure against an eavesdropper Eve, who observes the messages sent between Alice and Bob, but does not tamper with them. What if we replace Eve with Mallory, an active adversary (man-in-the-middle) who can tamper with messages?

It turns out the Diffie-Hellman key exchange protocol is only secure against a passive adversary and not an active adversary. If Mallory can tamper with the communication between Alice and Bob, she can fool them into thinking that they've agreed with a shared key, when they have actually generated two different keys that Mallory knows.

The following figure demonstrates how an active attacker (Mallory) can agree on a key ($K_1 = g^{am} \pmod p$) with Alice and another key ($K_2 = g^{bm} \pmod p$) with Bob in order to man-in-the-middle (MITM) their communications.

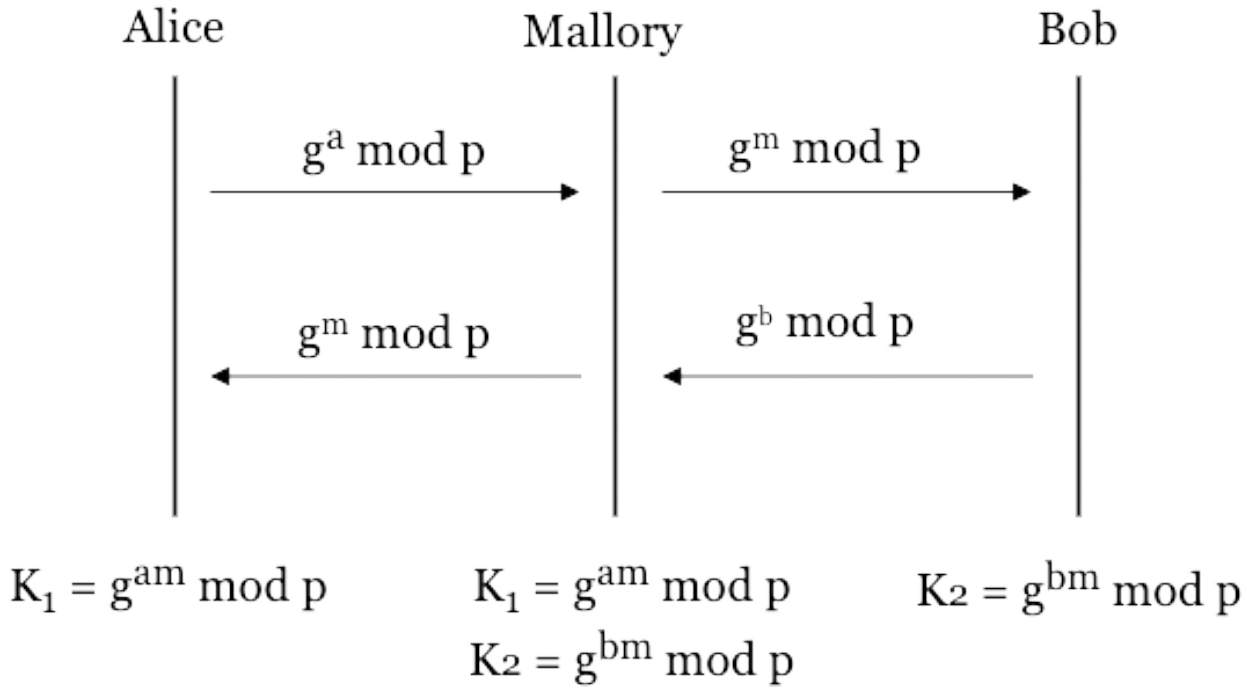

Figure 1: Diagram of the Diffie-Hellman key exchange between Alice and Bob, with Mallory in the middle

When Alice sends $g^a \pmod p$ to Bob, Mallory intercepts the message and replaces it with $g^m \pmod p$, where m is Mallory's secret. Bob now receives $g^m \pmod p$ instead of $g^a \pmod p$. Now, when Bob wants to calculate his shared key, he will calculate $K = A^b \pmod p$, where A is the value he received from Alice. Since he received a tampered value from Mallory, Bob will actually calculate $K = (g^m)^b = g^{mb} \pmod p$.

Likewise, when Bob sends $g^b \pmod p$ to Alice, Mallory intercepts the message and replaces it with $g^m \pmod p$. Alice receives $g^m \pmod p$. To calculate her shared key, she calculates $K = B^a \pmod p$, where B is the value she received from Bob. Since Alice received a tampered value, she will actually calculate $K = (g^m)^a = g^{ma} \pmod p$.

After the exchange, Alice thinks the shared key is $g^{ma} \pmod p$ and Bob thinks the shared key is $g^{mb} \pmod p$. They no longer have the same shared secret.

Even worse, Mallory knows both of these values too. Mallory intercepted Alice sending $g^a \pmod p$, which means Mallory knows the value of $g^a \pmod p$. She also knows her own chosen secret m. Thus she can calculate $(g^a)^m = g^{am} \pmod p$, which is what Alice thinks her shared secret is. Likewise, Mallory intercepted $g^b \pmod p$ from Bob and can calculate $(g^b)^m = g^{bm} \pmod p$, which is what Bob thinks his shared secret is.

If Alice and Bob fall victim to this attack, Mallory can now decrypt any messages sent from Alice with Alice's key $g^{ma} \pmod p$, make any changes to the message, re-encrypt the message with Bob's key $g^{mb} \pmod p$, and send it to Bob. In other words, Mallory would pretend to Alice that she is Bob, and pretend to Bob that she is Alice. This would not only allow Mallory to eavesdrop on the entire conversation but also make changes to the messages without Alice and Bob ever noticing that they are under attack.

The main reason why the Diffie-Hellman protocol is vulnerable to this attack is that the messages exchanged between Alice and Bob have no integrity or authenticity. To defend against this attack, Alice and Bob will need to additionally use a cryptoscheme that provides integrity and authenticity, such as digital signatures.

If the messages sent during the Diffie-Hellman exchange have integrity and authenticity, then Alice and Bob would be able to detect Mallory's tampering with the messages.

11. Public-Key Encryption

11. Public-Key (Asymmetric) Encryption

11.1. Overview

Previously we saw symmetric-key encryption, where Alice and Bob share a secret key K and use the same key to encrypt and decrypt messages. However, symmetric-key cryptography can be inconvenient to use, because it requires Alice and Bob to coordinate somehow and establish the shared secret key. *Asymmetric cryptography*, also known as *public-key cryptography*, is designed to address this problem.

In a public-key cryptosystem, the recipient Bob has a publicly available key, his *public key*, that everyone can access. When Alice wishes to send him a message, she uses his public key to encrypt her message. Bob also has a secret key, his *private key*, that lets him decrypt these messages. Bob publishes his public key but does not tell anyone his private key (not even Alice).

Public-key cryptography provides a nice way to help with the key management problem. Alice can pick a secret key K for some symmetric-key cryptosystem, then encrypt K under Bob's public key and send Bob the resulting ciphertext. Bob can decrypt using his private key and recover K. Then Alice and Bob can communicate using a symmetric-key cryptosystem, with K as their shared key, from there on.

11.2. Trapdoor One-way Functions

Public-key cryptography relies on a close variant of the one-way function. Recall from the previous section that a one-way function is a function f such that given x, it is easy to compute $f(x)$, but given y, it is hard to find a value x such that $f(x) = y$.

A *trapdoor one-way function* is a function f that is one-way, but also has a special backdoor that enables someone who knows the backdoor to invert the function. As before, given x, it is easy to compute $f(x)$, but given only y, it is hard to find a value x such that $f(x) = y$. However, given both y and the special backdoor K, it is now easy to compute x such that $f(x) = y$.

A trapdoor one-way function can be used to construct a public encryption scheme as follows. Bob has a public key PK and a secret key SK. He distributes PK to everyone, but does not share SK with anyone. We will use the trapdoor one-way function $f(x)$ as the encryption function.

Given the public key PK and a plaintext message x, it is computationally easy to compute the encryption of the message: $y = f(x)$.

Given a ciphertext y and only the public key PK, it is hard to find the plaintext message x where $f(x) = y$. However, given ciphertext y and the secret key SK, it becomes computationally easy to find the plaintext message x such that $y = f(x)$, i.e., it is easy to compute $f^{-1}(y)$.

We can view the private key as "unlocking" the trapdoor. Given the private key SK, it becomes easy to compute the decryption f^{-1}, and it remains easy to compute the encryption f.

Here are two examples of trapdoor functions that will help us build public encryption schemes:

- *RSA Hardness*: Suppose $n = pq$, i.e. n is the product of two large primes p and q. Given $c = m^e \pmod{n}$ and e, it is computationally hard to find m. However, with the factorization of n (i.e. p or q), it becomes easy to find m.

- *Discrete log problem*: Suppose p is a large prime and g is a generator. Given g, p, $A = g^a \pmod{p}$, and $B = g^b \pmod{p}$, it is computationally hard to find $g^{ab} \pmod{p}$. However, with a or b, it becomes easy to find $g^{ab} \pmod{p}$.

11.3. RSA Encryption

Under construction

For now, you can refer to these notes from CS 70 for a detailed proof of RSA encryption. For this class, you won't need to remember the proof of why RSA works. All you need to remember is that we use the public key to encrypt messages, we use the corresponding private key to decrypt messages, and an attacker cannot break RSA encryption unless they can factor large primes, which is believed to be hard.

There is a tricky flaw in the RSA scheme described in the CS 70 notes. The scheme is deterministic, so it is not IND-CPA secure. Sending the same message multiple times causes information leakage, because an adversary can see when the same message is sent. This basic variant of RSA might work for encrypting "random" messages, but it is not IND-CPA secure. As a result, we have to add some randomness to make the RSA scheme resistant to information leakage.

RSA introduces randomness into the scheme through a *padding mode*. Despite the name, RSA padding modes are more similar to the IVs in block cipher modes than the padding in block cipher modes. Unlike block cipher padding, public-key padding is not a deterministic algorithm for extending a message. Instead, public-key padding is a tool for mixing in some randomness so that the ciphertext output "looks random," but can still be decrypted to retrieve the original plaintext.

One common padding scheme is OAEP (Optimal Asymmetric Encryption Padding). This scheme effectively generates a random symmetric key, uses the random key to scramble the message, and encrypts both the scrambled message and the random key. To recover the original message, the attacker has to recover both the scrambled message and the random key in order to reverse the scrambling process.

11.4. El Gamal encryption

The Diffie-Hellman protocol doesn't quite deliver public-key encryption directly. It allows Alice and Bob to agree on a shared secret that they could use as a symmetric key, but it doesn't let Alice and Bob control what the shared secret is. For example, in the Diffie-Hellman protocol we saw, where Alice and Bob each choose random secrets, the shared secret is also a random value. Diffie-Hellman on its own does not let Alice and Bob send encrypted messages to each other. However, there is a slight variation on Diffie-Hellman that would allow Alice and Bob to exchange encrypted messages.

In 1985, a cryptographer by the name of Taher Elgamal invented a public-key encryption algorithm based on Diffie-Hellman. We will present a simplified form of El Gamal encryption scheme. El Gamal encryption works as follows.

The public system parameters are a large prime p and a value g satisfying $1 < g < p - 1$. Bob chooses a random value b (satisfying $0 \leq b \leq p - 2$) and computes $B = g^b \bmod p$. Bob's public key is B, and his private key is b. Bob publishes B to the world, and keeps b secret.

Now, suppose Alice has a message m (in the range $1 \ldots p - 1$) she wants to send to Bob, and suppose Alice knows that Bob's public key is B. To encrypt the message m to Bob, Alice picks a random value r (in the range $0 \ldots p - 2$), and forms the ciphertext

$(g^r \bmod p, m \times B^r \bmod p)$.

Note that the ciphertext is a pair of numbers, each number in the range $0 \ldots p - 1$.

How does Bob decrypt? Well, let's say that Bob receives a ciphertext of the form (R, S). To decrypt it, Bob computes

$R^{-b} \times S \bmod p$,

and the result is the message m Alice sent him.

Why does this decryption procedure work? If $R = g^r \bmod p$ and $S = m \times B^r \bmod p$ (as should be the case if Alice encrypted the message m properly), then

$R^{-b} \times S = (g^r)^{-b} \times (m \times B^r) = g^{-rb} \times m \times g^{br} = m \pmod p$.

If you squint your eyes just right, you might notice that El Gamal encryption is basically Diffie-Hellman, tweaked slightly. It's a Diffie-Hellman key exchange, where Bob uses his long-term public key B and where Alice uses a fresh new public key $R = g^r \bmod p$ chosen anew just for this exchange. They derive a shared key $K = g^{rb} = B^r = R^b \pmod p$. Then, Alice encrypts her message m by multiplying it by the shared key K modulo p.

That last step is in effect a funny kind of one-time pad, where we use multiplication modulo p instead of xor: here K is the key material for the one-time pad, and m is the message, and the ciphertext is $S = m \times K = m \times B^r \pmod p$. Since Alice chooses a new value r independently for each message she encrypts, we can see that the key material is indeed used only once. And a one-time pad using modular multiplication is just as secure as xor, for essentially the same reason that a one-time pad with xor is secure: given any ciphertext S and a hypothesized message m, there is exactly one key K that is consistent with this hypothesis (i.e., exactly one value of K satisfying $S = m \times K \bmod p$).

Another way you can view El Gamal is using the discrete log trapdoor one-way function defined above: Alice encrypts the message with $B^r = g^{br} \pmod p$. Given only g, p, $R = g^r \pmod p$, and $B = g^b \pmod p$, it is hard for an attacker to learn $g^{-br} \pmod p$ and decrypt the message. However, with Bob's secret key b, Bob can easily calculate $g^{-br} \pmod p$ and decrypt the message.

Note that for technical reasons that we won't go into, this simplified El Gamal scheme is actually *not* semantically secure. With some tweaks, the scheme can be made semantically secure. Interested readers can read more at this link.

Here is a summary of El Gamal encryption:

- **System parameters:** a 2048-bit prime p, and a value g in the range $2 \ldots p-2$. Both are arbitrary, fixed, and public.
- **Key generation:** Bob picks b in the range $0 \ldots p-2$ randomly, and computes $B = g^b \bmod p$. His public key is B and his private key is b.
- **Encryption:** $E_B(m) = (g^r \bmod p, m \times B^r \bmod p)$ where r is chosen randomly from $0 \ldots p-2$.
- **Decryption:** $D_b(R, S) = R^{-b} \times S \bmod p$.

11.5. Public Key Distribution

This all sounds great—almost too good to be true. We have a way for a pair of strangers who have never met each other in person to communicate securely with each other. Unfortunately, it is indeed too good to be true. There is a slight catch. The catch is that if Alice and Bob want to communicate securely using these public-key methods, they need some way to securely learn each others' public key. The algorithms presented here don't help Alice figure out what is Bob's public key; she's on her own for that.

You might think all Bob needs to do is broadcast his public key, for Alice's benefit. However, that's not secure against *active attacks*. Attila the attacker could broadcast his own public key, pretending to be Bob: he could send a spoofed broadcast message that appears to be from Bob, but that contains a public key that Attila generated. If Alice trustingly uses that public key to encrypt messages to Bob, then Attila will be able to intercept Alice's encrypted messages and decrypt them using the private key Attila chose.

What this illustrates is that Alice needs a way to obtain Bob's public key through some channel that she is confident cannot be tampered with. That channel does not need to protect the *confidentiality* of Bob's public key, but it does need to ensure the *integrity* of Bob's public key. It's a bit tricky to achieve this.

One possibility is for Alice and Bob to meet in person, in advance, and exchange public keys. Some computer security conferences have "key-signing parties" where like-minded security folks do just that. In a similar vein, some cryptographers print their public key on their business cards. However, this still requires Alice and Bob to meet in person in advance. Can we do any better? We'll soon see some methods that help somewhat with that problem.

11.6. Session Keys

There is a problem with public key: it is *slow*. It is very, very slow. When encrypting a single message with a 2048b RSA key, the RSA algorithm requires exponentiation of a 2048b number to a 2048b power, modulo a 2048b number. Additionally, some public key schemes only really work to encrypt "random" messages. For example, RSA without OAEP leaks when the same message is sent twice, so it is only secure if every message sent consists of random bits. In the simplified El Gamal scheme shown in these notes, it is easy for an attacker to substitute the message $M' = 2M$. If the messages have meaning, this can be a problem.

Because public key schemes are expensive and difficult to make IND-CPA secure, we tend to only use public key cryptography to distribute one or more *session keys*. Session keys are the keys used to actually encrypt and authenticate the message. To send a message, Alice first generates a random set of session keys. Often, we generate several different session keys for different purposes. For example, we may generate one key for encryption algorithms and another key for MAC algorithms. We may also generate one key to encrypt messages from Alice to Bob, and another key to encrypt messages from Bob to Alice. (If we need different keys for each message direction and different keys for encryption and MAC, we would need a total of four symmetric keys.) Alice then encrypts the message using a symmetric algorithm with the session keys (such as AES-128-CBC-HMAC-SHA-256 [1]) and encrypts the random session keys with Bob's public key. When he receives the ciphertext, Bob first decrypts the session keys and then uses the session keys to decrypt the original message.

[1] That is, using AES with 128b keys in CBC mode and then using HMAC with SHA-256 for integrity

12. Digital Signatures

12. Digital Signatures

We can use the ideas from public-key encryption to build asymmetric cryptographic schemes that guarantee integrity and authentication too. In this section, we will define *digital signatures*, which are essentially the public-key version of MACs, and show how they can help guarantee integrity and authentication.

12.1. Digital signature properties

Recall that in public-key encryption, anyone could use Bob's public key to encrypt a message and send it to him, but only Bob could use his secret key to decrypt the message. However, the situation is different for digital signatures. It would not really make sense if everyone could generate a signature on a message and only Bob could verify it. If anyone could generate a signature with a public key, what's stopping an attacker from generating a malicious message with a valid signature?

Instead, we want the reverse: only Bob can generate a signature on a message, and everyone else can verify the signature to confirm that the message came from Bob and has not been tampered with.

In a digital signature scheme, Bob generates a public key (also known as a *verification* key) and a private key (also known as a *signing* key). Bob distributes his public verification key to everyone, but keeps his signing key secret. When Bob wants to send a message, he uses his secret signing key to generate a signature on the message. When Alice receives the message, she can use Bob's public verification key to check that the signature is valid and confirm that the message is untampered and actually from Bob.

Mathematically, a digital signature scheme consists of three algorithms:

- **Key generation:** There is a randomized algorithm KeyGen that outputs a matching public key and private key: $(PK, SK) = \text{KeyGen}()$. Each invocation of KeyGen produces a new keypair.
- **Signing:** There is a signing algorithm Sign: $S = \text{Sign}(SK, M)$ is the signature on the message M (with private key SK).
- **Verification:** There is a verification algorithm Verify, where $\text{Verify}(PK, M, S)$ returns true if S is a valid signature on M (with public key PK) or false if not.

If PK, SK are a matching pair of private and public keys (i.e., they were output by some call to KeyGen), and if $S = \text{Sign}(SK, M)$, then $\text{Verify}(PK, M, S) = \text{true}$.

12.2. RSA Signatures: High-level Outline

At a high level, the RSA signature scheme works like this. It specifies a trapdoor one-way function F. The public key of the signature scheme is the public key U of the trapdoor function, and the private key of the signature scheme is the private key K of the trapdoor function. We also need a one-way function H, with no trapdoor; we typically let H be some cryptographic hash function. The function H is standardized and described in some public specification, so we can assume that everyone knows how to compute H, but no one knows how to invert it.

We define a signature on a message M as a value S that satisfies the following equation:

$H(M) = F_U(S)$.

Note that given a message M, an alleged signature S, and a public key U, we can verify whether it satisfies the above equation. This makes it possible to verify the validity of signatures.

How does the signer sign messages? It turns out that the trapdoor to F, i.e., the private key K, lets us find solutions to the above equation. Given a message M and the private key K, the signer can first compute $y = H(M)$, then find a value S such that $F_U(S) = y$. In other words, the signer computes $S = F^{-1}(H(M))$; that's the signature on M. This is easy to do for someone who knows the private key K, because K lets us invert the function F, but it is hard to do for anyone who does not know K. Consequently, anyone who has the private key can sign messages.

For someone who does not know the private key K, there is no easy way to find a message M and a valid signature S on it. For instance, an attacker could pick a message M, compute $H(M)$, but then the attacker would be unable to compute $F^{-1}(H(M))$, because the attacker does not know the trapdoor for the one-way function F. Similarly, an attacker could pick a signature S and compute $y = F(S)$, but then the attacker would be unable to find a message M satisfying $H(M) = y$, since H is one-way.

This is the general idea underpinning the RSA signature scheme. Now let's look at how to build a trapdoor one-way function, which is the key idea needed to make this all work.

12.3. Number Theory Background

Here are some basic facts from number theory, which will be useful in deriving RSA signatures. As previously discussed in lecture, we use $\varphi(n)$ to denote Euler's *totient function* of n: the number of positive integers less than n that share no common factor with n.

Fact 1 *If $gcd(x, n) = 1$, then $x^{\varphi(n)} = 1 \pmod{n}$. ("Euler's theorem.")*

Fact 2 *If p and q are two different odd primes, then $\varphi(pq) = (p-1)(q-1)$.*

Fact 3 *If $p = 2 \pmod 3$ and $q = 2 \pmod 3$, then there exists a number d satisfying $3d = 1 \pmod{\varphi(pq)}$, and this number d can be efficiently computed given $\varphi(pq)$.*

Let's assume that p and q are two different odd primes, that $p = 2 \pmod 3$ and $q = 2 \pmod 3$, and that $n = pq$.[1] Let d be the positive integer promised to exist by Fact 3. As a consequence of Facts 2 and 3, we can efficiently compute d given knowledge of p and q.

Theorem 1 *With notation as above, define functions F, G by $F(x) = x^3 \bmod n$ and $G(x) = x^d \bmod n$. Then $G(F(x)) = x$ for every x satisfying $gcd(x, n) = 1$.*

Proof: By Fact 3, $3d = 1 + k\varphi(n)$ for some integer k. Now applying Fact 1, we find

$G(F(x)) = (x^3)^d = x^{3d} = x^{1+k\varphi(n)} = x^1 \cdot (x^{\varphi(n)})^k = x \cdot 1^k = x \pmod{n}$.

The theorem follows.

If the primes p, q are chosen to be large enough—say, 1024-bit primes—then it is believed to be computationally infeasible to recover p and q from n. In other words, in these circumstances it is believed hard to factor the integer $n = pq$. It is also believed to be hard to recover d from n. And, given knowledge of only n (but not d or p, q), it is believed to be computationally infeasible to compute the function G. The security of RSA will rely upon this hardness assumption.

12.4. RSA Signatures

We're now ready to describe the RSA signature scheme. The idea is that the function F defined in Theorem 1 will be our trapdoor one-way function. The public key is the number n, and the private key is the number d. Given the public key n and a number x, anyone can compute $F(x) = x^3 \bmod n$. As mentioned before, F

[1] Why do we pick those particular conditions on p and q? Because then $\varphi(pq) = (p-1)(q-1)$ will not be a multiple of 3, which is going to allow us to have unique cube roots.

is (believed) one-way: given $y = x^3 \bmod n$, there is no known way to recover x in any reasonable amount of computing time. However, we can see that the private key d provides a trapdoor: given d and y, we can compute $x = G(y) = y^d \bmod n$. The intuition underlying this trapdoor function is simple: anyone can cube a number modulo n, but computing cube roots modulo n is believed to be hard if you don't know the factorization of n.

We then apply this trapdoor one-way function to the basic approach outlined earlier. Thus, a signature on message M is a value S satisfying

$H(M) = S^3 \bmod n$.

The RSA signature scheme is defined by the following three algorithms:

- **Key generation:** We can pick a pair of random 1024-bit primes p, q that are both 2 mod 3. Then the public key is $n = pq$, and the private key is the value of d given by Fact 3 (it can be computed efficiently using the extended Euclidean algorithm).
- **Signing:** The signing algorithm is given by

  $\text{Sign}_d(M) = H(M)^d \bmod n$.
- **Verification:** The verification algorithm Verify is given by

  $$\text{Verify}_n(M, S) = \begin{cases} \text{true} & \text{if } H(M) = S^3 \bmod n, \\ \text{false} & \text{otherwise.} \end{cases}$$

Theorem 1 ensures the correctness of the verification algorithm, i.e., that

$\text{Verify}_n(M, \text{Sign}_d(M)) = \text{true}$.

A quick reminder: in these notes we're developing the conceptual basis underlying MAC and digital signature algorithms that are widely used in practice, but again don't try to implement them yourself based upon just this discussion! We've omitted some technical details that do not change the big picture, but that are essential for security in practice. For your actual systems, use a reputable crypto library!

12.5. Definition of Security for Digital Signatures

Finally, let's outline a formal definition of what we mean when we say that a digital signature scheme is secure. The approach is very similar to what we saw for MACs.

We imagine a game played between Georgia (the adversary) and Reginald (the referee). Initially, Reginald runs KeyGen to get a keypair $\langle K, U \rangle$. Reginald sends the public key U to Georgia and keeps the private key K to himself. In each round of the game, Georgia may query Reginald with a message M_i; Reginald responds with $S_i = \text{Sign}_K(M_i)$. At any point, Georgia can yell "Bingo!" and output a pair $\langle M, S \rangle$. If this pair satisfies $\text{Verify}_U(M, S) = \text{true}$, and if Reginald has not been previously queried with the message M, then Georgia wins the game: she has forged a signature. Otherwise, Georgia loses.

If Georgia has any strategy to successfully forge a signature with non-negligible probability (say, with success probability at least $1/2^{40}$), given a generous amount of computation time (say, 2^{80} steps of computation) and any reasonable number of rounds of the game (say, 2^{40} rounds), then we declare the digital signature scheme insecure. Otherwise, we declare it secure.

This is a very stringent definition of security, because it declares the signature scheme broken if Georgia can successfully forge a signature on any message of her choice, even after tricking Alice into signing many messages of Georgia's choice. Nonetheless, modern digital signature algorithms—such as the RSA signature scheme—are believed to meet this definition of security.

Note however that the security of signatures do rely on the underlying hash function. Signatures have been broken in the past by taking advantage of the ability to create hash collisions when the hash function, not the public key algorithm, is compromised.

13. Certificates

13. Certificates

So far we've seen powerful techniques for securing communication such that the only information we must carefully protect regards "keys" of various sorts. Given the success of cryptography in general, arguably the biggest challenge remaining for its effective use concerns exactly those keys, and how to *manage* them. For instance, how does Alice find out Bob's public key? Does it matter?

13.1. Man-in-the-middle Attacks

Suppose Alice wants to communicate securely with Bob over an insecure communication channel, but she doesn't know his public key (and he doesn't know hers). A naive strategy is that she could just send Bob a message asking him to send his public key, and accept whatever response she gets back (over the insecure communication channel). Alice would then encrypt her message using the public key she received in this way.

This naive approach is insecure. An *active attacker* (Mallory, in our usual terminology) could tamper with Bob's response, replacing the public key in Bob's response with the attacker's public key. When Alice encrypts her message, she'll be encrypting it under Mallory's public key, not Bob's public key. When Alice transmits the resulting ciphertext over the insecure communication channel, Mallory can observe the ciphertext, decrypt it with his private key, and learn the secret message that Alice was trying to send to Bob.

You might think that Bob could detect this attack when he receives a ciphertext that he is unable to decrypt using his own private key. However, an active attacker can prevent Bob from noticing the attack. After decrypting the ciphertext Alice sent and learning the secret message that Alice wanted to send, Mallory can re-encrypt Alice's message under Bob's public key, though not before possibly tampering with Alice's packet to replace her ciphertext with new ciphertext of Mallory's choosing. In this way, neither Alice nor Bob would have any idea that something has gone wrong. This allows an active attacker to spy on—*and alter*—Alice's secret messages to Bob, without breaking any of the cryptography.

If Alice and Bob are having a two-way conversation, and they both exchange their public keys over an insecure communication channel, then Mallory can mount a similar attack in both directions. As a result, Mallory will get to observe all of the secret messages that Alice and Bob send to each other, but neither Alice nor Bob will have any idea that something has gone wrong. This is known as a "*man-in-the-middle*" (MITM) attack because the attacker interposes between Alice and Bob.

Man-in-the-middle attacks were possible in this example because Alice did not have any way of authenticating Bob's alleged public key. The general strategy for preventing MITM attacks is to ensure that every participant can verify the authenticity of other people's public keys. But how do we do that, specifically? We'll look next at several possible approaches to secure key management.

13.2. Trusted Directory Service

One natural approach to this key management problem is to use a trusted directory service: some organization that maintains an association between the name of each participant and their public key. Suppose everyone trusts Dirk the Director to maintain this association. Then any time Alice wants to communicate with someone, say Bob, she can contact Dirk to ask him for Bob's public key. This is only safe if Alice trusts

Dirk to respond correctly to those queries (e.g., not to lie to her, and to avoid being fooled by imposters pretending to be Bob): if Dirk is malicious or incompetent, Alice's security can be compromised.

On first thought, it sounds like a trusted directory service doesn't help, because it just pushes the problem around. If Alice communicates with the trusted directory service over an insecure communication channel, the entire scheme is insecure, because an active attacker can tamper with messages involving the directory service. To protect against this threat, Alice needs to know the directory service's public key, but where does she get *that* from? One potential answer might be to **hardcode** the public key of the directory service in the source code of all applications that rely upon the directory service. So this objection can be overcome.

A trusted directory service might sound like an appealing solution, but it has a number of shortcomings:

- *Trust:* It requires complete trust in the trusted directory service. Another way of putting this is that everyone's security is contingent upon the correct and honest operation of the directory service.
- *Scalability:* The directory service becomes a bottleneck. Everyone has to contact the directory service at the beginning of any communication with anyone new, so the directory service is going to be getting a lot of requests. It had better be able to answer requests very quickly, lest everyone's communications suffer.
- *Reliability:* The directory service becomes a single central point of failure. If it becomes unavailable, then no one can communicate with anyone not known to them. Moreover, the service becomes a single point of vulnerability to denial-of-service attacks: if an attacker can mount a successful DoS attack on the directory service, the effects will be felt globally.
- *Online:* Users will not be able to use this service while they are disconnected. If Alice is composing an email offline (say while traveling), and wants to encrypt it to Bob, her email client will not be able to look up Bob's public key and encrypt the email until she has connectivity again. As another example, suppose Bob and Alice are meeting in person in the same room, and Alice wants to use her phone to beam a file to Bob over infrared or Bluetooth. If she doesn't have general Internet connectivity, she's out of luck: she can't use the directory service to look up Bob's public key.
- *Security:* The directory service needs to be available in real time to answer these queries. That means that the machines running the directory service need to be Internet-connected at all times, so they will need to be carefully secured against remote attacks.

Because of these limitations, the trusted directory service concept is not widely used in practice, except in the context of messengers (such as Signal), where in order to send a message, Alice already has to be online.

In this case, the best approach is described as "trust but verify" using a key transparency mechanism. Suppose Alice and Bob discovered each others keys through the central keyserver. If they are ever in person, they can examine their devices to ensure that Alice actually has the correct key for Bob and vice versa. Although inconvenient, this acts as a check on a rogue keyserver, as the rogue keyserver would know there is at least a chance of getting caught.

However, some of these limitations—specifically, the ones relating to scalability, reliability, and the requirement for online access to the directory service—can be addressed through a clever idea known as digital certificates.

13.3. Digital Certificates

Digital certificates are a way to represent an alleged association between a person's name and their public key, as attested by some certifying party.

Let's look at an example. As a professor at UC Berkeley, David Wagner is an employee of the state of California. Suppose that the state maintained a list of each state employee's public key, to help Californians communicate with their government securely. The governor, Jerry Brown, might control a private key that is used to sign statements about the public key associated with each employee. For instance, Jerry could sign a statement attesting that "David Wagner's public key is `0x092...3F`", signed using the private key that Jerry controls.

In cryptographic protocol notation, the certificate would look like this:

Encryption under a public key: {David Wagner's public key is `0x092...3F`}$_{PK}$

Signing with private key: {David Wagner's public key is `0x092...3F`}$_{SK^{-1}_{\text{Jerry}}}$

where here $\{M\}_{SK^{-1}}$ denotes a digital signature on the message M using the private key SK^{-1}. In this case, SK^{-1}_{Jerry} is Jerry Brown's private key. This certificate is just some digital data: a sequence of bits. The certificate can be published and shared with anyone who wants to communicate securely with David.

If Alice wants to communicate securely with David, she can obtain a copy of this certificate. If Alice knows Jerry's public key, she can verify the signature on David's digital certificate. This gives her high confidence that indeed Jerry consented to the statement about the bit pattern of David's public key, because the valid signature required Jerry to decide to agree to apply his private key to the statement.

If Alice also considers Jerry trustworthy and competent at recording the association between state employees and their public keys, she can then conclude that David Wagner's public key is `0x092...3F`, and she can use this public key to securely communicate with David.

Notice that Alice did not need to contact a trusted directory service. She only needed to receive a copy of the digital certificate, but she could obtain it from *anyone*—by Googling it, by obtaining it from an untrusted directory service, by seeing it scrawled on a whiteboard, or by getting a copy from David himself. It's perfectly safe for Alice to download a copy of the certificate over an insecure channel, or to obtain it from an untrustworthy source, as long as she verifies the signature on the digital certificate and trusts Jerry for these purposes. The certificate is, in some sense, self-validating. Alice has *bootstrapped* her trust in the validity of David's public key based on her existing trust that she has a correct copy of Jerry's public key, *plus* her belief that Jerry takes the act of signing keys seriously, and won't sign a statement regarding David's public key unless Jerry is sure of the statement's correctness.

13.4. Public-Key Infrastructure (PKI)

Let's now put together the pieces. A *Certificate Authority* (CA) is a party who issues certificates. If Alice trusts some CA, and that CA issues Bob a digital certificate, she can use Bob's certificate to get a copy of Bob's public key and securely communicate with him. For instance, in the example of the previous section, Jerry Brown acted as a CA for all employees of the state of California.

In general, if we can identify a party who everyone in the world trusts to behave honestly and competently—who will verify everyone's identity, record their public key accurately, and issue a public certificate to that person accordingly—that party can play the role of a trusted CA. The public key of the trusted CA can be hardcoded in applications that need to use cryptography. Whenever an application needs to look up David Wagner's public key, it can ask David for a copy of his digital certificate, verify that it was properly signed by the trusted CA, extract David's public key, and then communicate securely with David using his public key.

Some of the criticisms of the trusted directory service mentioned earlier also apply to this use of CAs. For instance, the CA must be trusted by everyone: put another way, Alice's security can be breached if the CA behaves maliciously, makes a mistake, or acts without sufficient care. So we need to find a single entity whom everyone in the world can agree to trust—a tall order. However, digital certificates have better scalability, reliability, and utility than an online directory service.

For this reason, digital certificates are widely used in practice today, with large companies (e.g., Verisign) having thriving businesses acting as CAs.

This model is also used to secure the web. A web site that wishes to offer access via SSL (`https:`) can buy a digital certificate from a CA, who checks the identity of the web site and issues a certificate linking the site's domain name (e.g., `www.amazon.com`) to its public key. Every browser in the world ships with a list of trusted CAs. When you type in an `https:` URL into your web browser, it connects to the web site, asks for a copy of the site's digital certificate, verifies the certificate using the public key of the CA who issued it, checks that the domain name in the certificate matches the site that you asked to visit, and then establishes secure communications with that site using the public key in the digital certificate.

Web browsers come configured with a list of many trusted CAs. As a fun exercise, you might try listing the set of trusted CAs configured in your web browser and seeing how many of the names you can recognize. If you use Firefox, you can find this list by going to Preferences / Advanced / Certificates / View Certificates / Authorities. Firefox currently ships with about 88 trusted CAs preconfigured in the browser. Take a look and see what you think of those CAs. Do you know who those CAs are? Would you consider them trustworthy? You'll probably find many unfamiliar names. For instance, who is Unizeto? TURKTRUST? AC Camerfirma? XRamp Security Services? Microsec Ltd? Dhimyotis? Chunghwa Telecom Co.? Do you trust them?

The browser manufacturers have decided that, whether you like it or not, those CAs are trusted. You might think that it's an advantage to have many CAs configured into your browser, because that gives each user a choice depending upon whom they trust. However, that's not how web browsers work today. Your web browser will accept *any* certificate issued by *any* of these 88 CAs. If Dhimyotis issues a certificate for `amazon.com`, your browser will accept it. Same goes for all the rest of your CAs. This means that if any one of those 88 CAs issues a certificate to the wrong person, or behaves maliciously, that could affect the security of everyone who uses the web. The more CAs your browser trusts, the greater the risk of a security breach. That CA model is under increasing criticism for these reasons.

13.5. Certificate Chains and Hierarchical PKI

Above we looked at an example where Jerry Brown could sign certificates attesting to the public keys of every California state employee. However, in practice that may not be realistic. There are over 200,000 California state employees, and Jerry couldn't possibly know every one of them personally. Even if Jerry spent all day signing certificates, he still wouldn't be able to keep up—let alone serve as governor.

A more scalable approach is to establish a hierarchy of responsibility. Jerry might issue certificates to the heads of each of the major state agencies. For instance, Jerry might issue a certificate for the University of California, delegating to UC President Janet Napolitano the responsibility and authority to issue certificates to UC employees. Napolitano might sign certificates for all UC employees. We get:

$\{\text{The University of California's public key is } PK_{\text{Napolitano}}\}_{SK^{-1}_{\text{Jerry}}}$

$\{\text{David Wagner's public key is } PK_{\text{daw}}\}_{SK^{-1}_{\text{Napolitano}}}$

This is a simple example of a *certificate chain*: a sequence of certificates, each of which authenticates the public key of the party who has signed the next certificate in the chain.

Of course, it might not be realistic for President Napolitano to personally sign the certificates of all UC employees. We can imagine more elaborate and scalable scenarios. Jerry might issue a certificate for UC to Janet Napolitano; Napolitano might issue a certificate for UC Berkeley to UCB Chancellor Nicholas Dirks; Dirks might issue a certificate for the UCB EECS department to EECS Chair Randy Katz; and Katz might issue each EECS professor a certificate that attests to their name, public key, and status as a state employee. This would lead to a certificate chain of length 4.

In the latter example, Jerry acts as a Certificate Authority (CA) who is the authoritative source of information about the public key of each state agency; Napolitano serves as a CA who manages the association between UC campuses and public keys; Dirks serves as a CA who is authoritative regarding the public key of each UCB department; and so on. Put another way, Jerry delegates the power to issue certificates for UC employees to Napolitano; Napolitano further sub-delegates this power, authorizing Dirks to control the association between UCB employees and their public keys; and so on.

In general, the hierarchy forms a tree. The depth can be arbitrary, and thus certificate chains may be of any length. The CA hierarchy is often chosen to reflect organizational structures.

13.6. Revocation

What do we do if a CA issues a certificate in error, and then wants to invalidate the certificate? With the basic approach described above, there is nothing that can be done: a certificate, once issued, remains valid forever.

This problem has arisen in practice. A number of years ago, Verisign issued bogus certificates for "Microsoft Corporation" to ... someone other than Microsoft. It turned out that Verisign had no way to revoke those bogus certificates. This was a serious security breach, because it provided the person who received those certificates with the ability to run software with all the privileges that would be accorded to the real Microsoft. How was this problem finally resolved? In the end, Microsoft issued a special patch to the Windows operating system that revoked those specific bogus certificates. The patch contained a hardcoded copy of the bogus certificates and inserted an extra check into the certificate-checking code: if the certificate matches one of the bogus certificates, then treat it as invalid. This addressed the particular issue, but was only feasible because Microsoft was in a special position to push out software to address the problem. What would we have done if a trusted CA had handed out a bogus certificate for Amazon.com, or Paypal.com, or BankofAmerica.com, instead of for Microsoft.com?

This example illustrates the need to consider revocation when designing a PKI system. There are two standard approaches to revocation:

- *Validity periods.* Certificates can contain an expiration date, so they're no longer considered valid after the expiration date. This doesn't let you immediately revoke a certificate the instant you discover that it was issued in error, but it limits the damage by ensuring that the erroneous certificate will eventually expire.

  With this approach, there is a fundamental tradeoff between efficiency and how quickly one can revoke an erroneous certificate. On the one hand, if the lifetime of each certificate is very short—say, each certificate is only valid for a single day, and then you must request a new one—then we have a way to respond quickly to bad certificates: a bad certificate will circulate for at most one day after we discover it. Since we won't re-issue certificates known to be bad, after the lifetime elapses the certificate has effectively been revoked. However, the problem with short lifetimes is that legitimate parties must frequently contact their CA to get new certificates; this puts a heavy load on all the parties, and can create reliability problems if the CA is unreachable for a day. On the other hand, if we set the lifetime very long, then reliability problems can be avoided and the system scales well, but we lose the ability to respond promptly to erroneously issued certificates.

- *Revocation lists.* Alternatively, the CA could maintain and publish a list of all certificates it has revoked. For security, the CA could date and digitally sign this list. Every so often, everyone could download the latest copy of this revocation list, check its digital signature, and cache it locally. Then, when checking the validity of a digital certificate, we also check that it is not on our local copy of the revocation list.

  The advantage of this approach is that it offers the ability to respond promptly to bad certificates. There is a tradeoff between efficiency and prompt response: the more frequently we ask everyone to download the list, the greater the load on the bandwidth and on the CA's revocation servers, but the more quickly we can revoke bad certificates. If revocation is rare, this list might be relatively short, so revocation lists have the potential to be more efficient than constantly re-issuing certificates with a short validity period.

  However, revocation lists also pose some special challenges of their own. What should clients do if they are unable to download a recent copy of the revocation list? If clients continue to use an old copy of the revocation list, then this creates an opportunity for an attacker who receives a bogus certificate to DoS the CA's revocation servers in order to prevent revocation of the bogus certificate. If clients err on the safe side by rejecting all certificates if they cannot download a recent copy of the revocation list, this creates an even worse problem: an attacker who successfully mounts a sustained DoS attack on the CA's revocation servers may be able to successfully deny service to all users of the network.

  Today, systems that use revocation lists typically ignore these denial-of-service risks and hope for the best.

13.7. Web of Trust

Another approach is the so-called *web of trust*, which was pioneered by PGP, a software package for email encryption. The idea is to democratize the process of public key verification so that it does not rely upon

any single central trusted authority. In this approach, each person can issue certificates for their friends, colleagues, and others whom they know.

Suppose Alice wants to contact Doug, but she doesn't know Doug. In the simplest case, if she can find someone she knows and trusts who has issued Doug a certificate, then she has a certificate for Doug, and everything is easy.

If that doesn't work, things get more interesting. Suppose Alice knows and trusts Bob, who has issued a certificate to Carol, who has in turn issued a certificate to Doug. In this case, PGP will use this certificate chain to identify Doug's public key.

In the latter scenario, is this a reasonable way for Alice to securely obtain a copy of Doug's public key? It's hard to say. For example, Bob might have carefully checked Carol's identity before issuing her a certificate, but that doesn't necessarily indicate how careful or honest Carol will be in signing other people's keys. In other words, Bob's signature on the certificate for Carol might attest to Carol's *identity*, but not necessarily her honesty, integrity, or competence. If Carol is sloppy or malicious, she might sign a certificate that purports to identify Doug's public key, but actually contains some imposter's public key instead of Doug's public key. That would be bad.

This example illustrates two challenges:

- *Trust isn't transitive.* Just because Alice trusts Bob, and Bob trusts Carol, it doesn't necessarily follow that Alice trusts Carol. (More precisely: Alice might consider Bob trustworthy, and Bob might consider Carol trustworthy, but Alice might not consider Carol trustworthy.)
- *Trust isn't absolute.* We often trust a person for a specific purpose, without necessarily placing absolute trust in them. To quote one security expert: "I trust my bank with my money but not with my children; I trust my relatives with my children but not with my money." Similarly, Alice might trust that Bob will not deliberately act with malicious intent, but it's another question whether Alice trusts Bob to very diligently check the identity of everyone whose certificate he signs; and it's yet another question entirely whether Alice trusts Bob to have good judgement about whether third parties are trustworthy.

The web-of-trust model doesn't capture these two facets of human behavior very well.

The PGP software takes the web of trust a bit further. PGP certificate servers store these certificates and make it easier to find an intermediary who can help you in this way. PGP then tries to find *multiple* paths from the sender to the recipient. The idea is that the more paths we find, and the shorter they are, the greater the trust we can have in the resulting public key. It's not clear, however, whether there is any principled basis for this theory, or whether this really addresses the issues raised above.

One criticism of the web-of-trust approach is that, empirically, many users find it hard to understand. Most users are not experts in cryptography, and it remains to be seen whether the web of trust can be made to work well for non-experts. To date, the track record has not been one of strong success. Even in the security community, it is only partially used—not due to lack of understanding, but due to usability hurdles, including lack of integration into mainstream tools such as mail readers.

13.8. Leap-of-Faith Authentication

Another approach to managing keys is exemplified by SSH. The first time that you use SSH to connect to a server you've never connected to before, your SSH client asks the server for its public key, the server responds in the clear, and the client takes a "leap of faith" and trustingly accepts whatever public key it receives.[1] The client remembers the public key it received from this server. When the client later connects to the same server, it uses the same public key that it obtained during the first interaction.

This is known as *leap-of-faith authentication*[2] because the client just takes it on faith that there is no man-in-the-middle attacker the first time it connects to the server. It has also sometimes been called *key*

[1] The client generally asks the user to confirm the trust decision, but users almost always ok the leap-of-faith.
[2] Another term is TOFU = Trust On First Use.

continuity management, because the approach is to ensure that the public key associated with any particular server remains unchanged over a long time period.

What do you think of this approach?

- A rigorous cryptographer might say: this is totally insecure, because an attacker could just mount a MITM attack on the first interaction between the client and server.
- A pragmatist might say: that's true, but it still prevents many kinds of attacks. It prevents passive eavesdropping. Also, it defends against any attacker who wasn't present during the first interaction, and that's a significant gain.
- A user might say: this is easy to use. Users don't need to understand anything about public keys, key management, digital certificates or other cryptographic concepts. Instead, the SSH client takes care of security for them, without their involvement. The security is invisible and automatic.

Key continuity management exemplifies several design principles for "usable security". One principle is that "there should be only one mode of operation, and it should be secure." In other words, users should not have to configure their software specially to be secure. Also, users should not have to take an explicit step to enable security protections; the security should be ever-present and enabled automatically, in all cases. Arguably, users should not even have the power to disable the security protections, because that opens up the risk of social engineering attacks, where the attacker tries to persuade the user to turn off the cryptography.

Another design principle: "Users shouldn't have to understand cryptography to use the system securely." While it's reasonable to ask the designers of the system to understand cryptographic concepts, it is not reasonable to expect users to know anything about cryptography.

14. Passwords

14. Passwords

Passwords are widely used for authentication, especially on the web. What practices should be used to make passwords as secure as possible?

14.1. Risks and weaknesses of passwords

Passwords have some well-known usability shortcomings. Security experts recommend that people pick long, strong passwords, but long random passwords are harder to remember. In practice, users are more likely to choose memorable passwords, which may be easier to guess. Also, rather than using a different, independently chosen password for each site, users often reuse passwords across multiple sites, for ease of memorization. This has security consequences as well.

From a security perspective, we can identify a number of security risks associated with password authentication:

- *Online guessing attacks.* An attacker could repeatedly try logging in with many different guesses at the user's password. If the user's password is easy to guess, such an attack might succeed.
- *Social engineering and phishing.* An attacker might be able to fool the user into revealing his/her password, e.g., on a phishing site. We've examined this topic previously, so we won't consider it further in these notes.
- *Eavesdropping.* Passwords are often sent in cleartext from the user to the website. If the attacker can eavesdrop (e.g., if the user is connecting to the Internet over an open Wifi network), and if the web connection is not encrypted, the attacker can learn the user's password.
- *Client-side malware.* If the user has a keylogger or other client-side malware on his/her machine, the keylogger/malware can capture the user's password and exfiltrate it to the attacker.
- *Server compromise.* If the server is compromised, an attacker may be able to learn the passwords of people who have accounts on that site. This may help the attacker break into their accounts on other sites.

We'll look at defenses and mitigations for each of these risks, below.

14.2. Mitigations for eavesdropping

There is a straightforward defense against eavesdropping: we can use SSL (also known as TLS). In other words, instead of connecting to the web site via http, the connection can be made over https. This will ensure that the username and password are sent over an encrypted channel, so an eavesdropper cannot learn the user's password.

Today, many sites do use SSL, but many do not.

Another possible defense would be to use more advanced cryptographic protocols. For instance, one could imagine a challenge-response protocol where the server sends your browser a random challenge r; then the browser takes the user's password w, computes $H(w, r)$ where H is a cryptographic hash (e.g., SHA256), and sends the result to the server. In this scheme, the user's password never leaves the browser and is never

sent over the network, which defends against eavesdroppers. Such a scheme could be implemented today with Javascript on the login page, but it has little or no advantage over SSL (and it has some shortcomings compared to using SSL), so the standard defense is to simply use SSL.

14.3. Mitigations for client-side malware

It is very difficult to protect against client-side malware.

To defend against keyloggers, some people have proposed using randomized virtual keyboards: a keyboard is displayed on the screen, with the order of letters and numbers randomly permuted, and the user is asked to click on the characters of their password. This way, a keylogger (which only logs the key strokes you enter) would not learn your password. However, it is easy for malware to defeat this scheme: for instance, the malware could simply record the location of each mouse click and take a screen shot each time you click the mouse.

In practice, if you type your password into your computer and your computer has malware on it, then the attacker learns your password. It is hard to defend against this; passwords are fundamentally insecure in this threat model. The main defense is two-factor authentication, where we combine the password with some other form of authentication (e.g., a SMS sent to your phone).

14.4. Online guessing attacks

How easy are online guessing attacks? Researchers have studied the statistics of passwords as used in the field, and the results suggest that online guessing attacks are a realistic threat. According to one source, the five most commonly used passwords are `123456`, `password`, `12345678`, `qwerty`, `abc123`. Of course, a smart attacker will start by guessing the most likely possibilities for the password first before moving on to less likely possibilities. A careful measurement study found that with a dictionary of the 10 most common passwords, you can expect to find about 1% of users' passwords. In other words, about 1% of users choose a password from among the top 10 most commonly used passwords. It also found that, with a dictionary of the 2^{20} most commonly used passwords, you can expect to guess about 50% of users' passwords: about half of all users will have a password that is in that dictionary.

One implication is that, if there are no limits on how many guesses an attacker is allowed to make, an attacker can have a good chance of guessing a user's password correctly. We can distinguish targeted from untargeted attacks. A *targeted attack* is where the attacker has a particular target user in mind and wants to learn their password; an *untargeted attack* is where the attacker just wants to guess some user's password, but doesn't care which user gets hacked. An untargeted attack, for instance, might be relevant if the attacker wants to take over some existing Gmail account and send lots of spam from it.

The statistics above let us estimate the work an attacker would have to do in each of these attack settings. For an untargeted attack, the attacker might try 10 guesses at the password against each of a large list of accounts. The attacker can expect to have to try about 100 accounts, and thus make a total of about 1000 login attempts, to guess one user's password correctly. Since the process of guessing a password and seeing if it is correct can be automated, resistance against untargeted attacks is very low, given how users tend to choose their passwords in practice.

For a targeted attack, the attacker's workload has more variance. If the attacker is extremely lucky, he might succeed within the first 10 guesses (happens 1% of the time). If the attacker is mildly lucky, he might succeed after about one million guesses (happens half of the time). If the attacker is unlucky, it might take a lot more than one million guesses. If each attempt takes 1 second (to send the request to the server and wait for the response), making 2^{20} guesses will take about 11 days, and the attack is very noticeable (easily detectable by the server). So, targeted attacks are possible, but the attacker is not guaranteed a success, and it might take quite a few attempts.

14.5. Mitigations for online guessing attacks

Let's explore some possible mitigations for online guessing:

- *Rate-limiting.* We could impose a limit on the number of consecutive incorrect guesses that can be made; if that limit is exceeded, the account is locked and the user must do something extra to log in (e.g., call up customer service). Or, we can impose a limit on the maximum guessing rate; if the number of incorrect guesses exceeds, say, 5 per hour, then we temporarily lock the account or impose a delay before the next attempt can be made.

  Rate-limiting is a plausible defense against targeted attacks. It does have one potential disadvantage: it introduces the opportunity for denial-of-service attacks. If Mallory wants to cause Bob some grief, Mallory can make enough incorrect login attempts to cause Bob's account to be locked. In many settings, though, this denial-of-service risk is acceptable. For instance, if we can limit each account to 5 incorrect guesses per hour, making 2^{20} guesses would take at least 24 years—so at least half of our user population will become essentially immune to targeted attacks.

  Unfortunately, rate-limiting is not an effective defense against untargeted attacks. An attacker who can make 5 guesses against each of 200 accounts (or 1 guess against each of 1000 accounts) can expect to break into at least one of them. Rate-limiting probably won't prevent the attacker from making 5 guesses (let alone 1 guess).

  Even with all of these caveats, rate-limiting is probably a good idea. Unfortunately, one research study found that only about 20% of major web sites currently use rate-limiting.

- *CAPTCHAs.* Another approach could be to try to make it harder to perform *automated* online guessing attacks. For instance, if a login attempt for some user fails, the system could require that the next time you try to log into that same account, you have to solve a CAPTCHA. Thus, making n guesses at the password for a particular user would require solving $n - 1$ CAPTCHAs. CAPTCHAs are designed to be solvable for humans but (we hope) not for computers, so we might hope that this would eliminate automated/scripted attacks.

  Unfortunately, this defense is not as strong as we might hope. There are black-market services which will solve CAPTCHAs for you. They even provide easy-to-use APIs and libraries so you can automate the process of getting the solution to the CAPTCHA. These services employ human workers in countries with low wages to solve the CAPTCHAs. The market rate is about \$1–2 per thousand CAPTCHAs solved, or about 0.1–0.2 cents per CAPTCHA solved. This does increase the cost of a targeted attack, but not beyond the realm of possibility.

  CAPTCHAs do not stop an untargeted attack. For instance, an attacker who makes one guess at each of 1000 accounts won't have to solve any CAPTCHAs. Or, if for some reason the attacker wants to make 10 guesses at each of 100 accounts, the attacker will only have to solve 900 CAPTCHAs, which will cost the attacker maybe a dollar or two: not very much.

- *Password requirements or nudges.* A site could also impose password requirements (e.g., your password must be 10 characters long and contain at least 1 number and 1 punctuation symbol). However, these requirements offer poor usability, are frustrating for users, and may just tempt some users to evade or circumvent the restriction, thus not helping security. Therefore, I would be reluctant to recommend stringent password requirements, except possibly in special cases.

  Another approach is to apply a gentle "nudge" rather than impose a hard requirement. For instance, studies have found that merely showing a password meter during account creation can help encourage people to choose longer and stronger passwords.

14.6. Mitigations for server compromise

The natural way to implement password authentication is for the website to store the passwords of all of its passwords in the clear, in its database. Unfortunately, this practice is bad for security. If the site gets hacked and the attacker downloads a copy of the database, then now all of the passwords are breached; recovery may be painful. Even worse, because users often reuse their passwords on multiple sites, such a security breach may now make it easier for the attacker to break into the user's accounts on other websites.

For these reasons, security experts recommend that sites avoid storing passwords in the clear. Unfortunately, sites don't always follow this advice. For example, in 2009, the Rockyou social network got hacked, and the hackers stole the passwords of all 32 million of their users and posted them on the Internet; not good. One study estimates that about 30–40% of sites still store passwords in the clear.

14.7. Password hashing

If storing passwords in the clear is not a good idea, what can we do that is better? One simple approach is to hash each password with a cryptographic hash function (say, SHA256), and store the hash value (not the password) in the database.

In more detail, when Alice creates her account and enters her password w, the system can hash w to get $H(w)$ and store $H(w)$ in the user database. When Alice returns and attempts to log in, she provides a password, say w'; the system can check whether this is correct by computing the hash $H(w')$ of w' and checking whether $H(w')$ matches what is in the user database.

Notice that the properties of cryptographic hash functions are very convenient for this application. Because cryptographic hash functions are one-way, it should be hard to recover the password w from the hash $H(w)$; so if there is a security breach and the attacker steals a copy of the database, no cleartext passwords are revealed, and it should be hard for the attacker to invert the hash and find the user's hashes. That's the idea, anyway.

Unfortunately, this simple idea has some shortcomings:

- *Offline password guessing.* Suppose that Mallory breaks into the website and steals a copy of the password database, so she now has the SHA256 hash of Bob's password. This enables her to test guesses at Bob's password very quickly, on her own computer, without needing any further interaction with the website. In particular, given a guess g at the password, she can simply hash g to get $H(g)$ and then test whether $H(g)$ matches the password hash in the database. By using lists of common passwords, English words, passwords revealed in security breaches of sites who didn't use password hashing, and other techniques, one can generate many guesses. This is known as an *offline guessing attack*: offline, because Mallory doesn't need to interact with the website to test a guess at the password, but can check her guess entirely locally.

  Unfortunately for us, a cryptographic hash function like SHA256 is very fast. This lets Mallory test many guesses rapidly. For instance, on modern hardware, it is possible to test something in the vicinity of 1 billion passwords per second (i.e., to compute about 1 billion SHA256 hashes per second). So, imagine that Mallory breaks into a site with 100 million users. Then, by testing 2^{20} guesses at each user's password, she can learn about half of those users' passwords. How long will this take? Well, Mallory will need to make 100 million $\times 2^{20}$ guesses, or a total of about 100 trillion guesses. At 1 billion guesses per second, that's about a day of computation. Ouch. In short, the hashing of the passwords helps some, but it didn't help nearly as much as we might have hoped.

- *Amortized guessing attacks.* Even worse, the attack above can be sped up dramatically by a more clever algorithm that avoids unnecessarily repeating work. Notice that we're going to try guessing the same 2^{20} plausible passwords against each of the users. And, notice that the password hash $H(w)$ doesn't depend upon the user: if Alice and Bob both have the same password, they'll end up with the same password hash.

  So, consider the following optimized algorithm for offline password guessing. We compute a list of 2^{20} pairs $(H(g), g)$, one for each of the 2^{20} most common passwords g, and sort this list by the hash value. Now, for each user in the user database, we check to see whether their password hash $H(w)$ is in the sorted list. If it is in the list, then we've immediately learned that user's password. Checking whether their password hash is in the sorted list can be done using binary search, so it can be done extremely efficiently (with about $\lg 2^{20} = 20$ random accesses into the sorted list). The attack requires computing 2^{20} hashes (which takes about one millisecond), sorting the list (which takes fractions of a second), and doing 100 million binary searches (which can probably be done in seconds or minutes, in total). This is

much faster than the previous offline guessing attack, because we avoid repeated work: we only need to compute the hash of each candidate password once.

14.8. Password hashing, done right

With these shortcomings in mind, we can now identify a better way to store passwords on the server.

First, we can eliminate the amortized guessing attack by *incorporating randomness into the hashing process.* When we create a new account for some user, we pick a random *salt* s. The salt is a value whose only purpose is to be different for each user; it doesn't need to be secret. The password hash for password w is $H(w, s)$. Notice that the password hash depends on the salt, so even if Alice and Bob share the same password w, they will likely end up with different hashes (Alice will have $H(w, s_A)$ and Bob $H(w, s_B)$, where most likely $s_A \neq s_B$). Also, to enable the server to authenticate each user in the future, the salt for each user is stored in the user database.

Instead of storing $H(w)$ in the database, we store $s, H(w, s)$ in the database, where s is a random salt. Notice that s is stored in cleartext, so if the attacker gets a copy of this database, the attacker will see the value of s. That's OK; the main point is that each user will have a different salt, so the attacker can no longer use the amortized guessing attack above. For instance, if the salt for Alice is s_A, the attacker can try guesses $g_1, g_2, \ldots, g_n$ at her password by computing $H(g_1, s_A), \ldots, H(g_n, s_A)$ and comparing each one against her password hash $H(w_A, s_A)$. But now when the attacker wants to guess Bob's password, he can't reuse any of that computation; he'll need to compute a new, different set of hashes, i.e., $H(g_1, s_B), \ldots, H(g_n, s_B)$, where s_B is the salt for Bob.

Salting is good, because it increases the attacker's workload to invert many password hashes. However, it is not enough. As the back-of-the-envelope calculation above illustrated, an attacker might still be able to try 2^{20} guesses at the password against each of 100 million users' password hashes in about a day. That's not enough to prevent attacks. For instance, when LinkedIn had a security breach that exposed the password hashes of all of their users, it was discovered that they were using SHA256, and consequently one researcher was able to recover 90% of their users' passwords in just 6 days. Not good.

So, the second improvement is to *use a slow hash.* The reason that offline password guessing is so efficient is because SHA256 is so fast. If we had a cryptographic hash that was very slow—say, it took 1 millisecond to compute—then offline password guessing would be much slower; an attacker could only try 1000 guesses at the password per second.

One way to take a fast hash function and make it slower is by iterating it. In other words, if H is a cryptographic hash function like SHA256, define the function F by

$$F(x) = H(H(H(\cdots(H(x))\cdots))),$$

where we have iteratively applied H n times. Now F is a good cryptographic hash function, and evaluating F will be n times slower than evaluating H. This gives us a tunable parameter that lets us choose just how slow we want the hash function to be.

Therefore, our final construction is to store $s, F(w, s)$ in the database, where s is a randomly chosen salt, and F is a slow hash constructed as above. In other words, we store

$$s, H(H(H(\cdots(H(w, s))\cdots)))$$

in the database.

How slow should the hash function F be? In other words, how should we choose n? On the one hand, for security, we'd like n to be as large as possible: the larger it is, the slower offline password guessing will be. On the other hand, we can't make it too large, because that will slow down the legitimate server: each time a user tries to log in, the server needs to evaluate F on the password that was provided. With these two considerations, we can now choose the parameter n to provide as much security as possible while keeping the performance overhead of slow hashing down to something unnoticeable.

For instance, suppose we have a site that expects to see at most 10 logins per second (that would be a pretty high-traffic site). Then we could choose n so that evaluating F takes about one millisecond. Now the legitimate server can expect to spend 1% of its CPU power on performing password hashes—a small performance hit. The benefit is that, if the server should be compromised, offline password guessing attacks will take the attacker a lot longer. With the example parameters above, instead of taking 1 day to try 2^{20} candidate passwords against all 100 million users, it might take the attacker about 3000 machine-years. That's a real improvement.

In practice, there are several existing schemes for slow hashing that you can use: Scrypt, Bcrypt, or PBKDF2. They all use some variant of the "iterated hashing" trick mentioned above.

14.9. Implications for cryptography

The analysis above has implications for the use of human-memorable passwords or passphrases for cryptography.

Suppose we're building a file encryption tool. It is tempting to prompt the user to enter in a password w, hash it using a cryptographic hash function (e.g., SHA256), use $k = H(w)$ as a symmetric key, and encrypt the file under k. Unfortunately, this has poor security. An attacker could try the 2^{20} most common passwords, hash each one, try decrypting under that key, and see if the decryption looks plausibly like plaintext. Since SHA256 is fast, this attack will be very fast, say one millisecond; and based upon the statistics mentioned above, this attack might succeed half of the time or so.

You can do a little bit better if you use a slow hash to generate the key instead of SHA256. Unfortunately, this isn't enough to get strong security. For example, suppose we use a slow hash tuned to take 1 millisecond to compute the hash function. Then the attacker can make 1000 guesses per second, and it'll take only about 15 minutes to try all 2^{20} most likely passwords; 15 minutes to have a 50% chance of breaking the crypto doesn't sound so hot.

The unavoidable conclusion is that deriving cryptographic keys from passwords, passphrases, or human-memorable secrets is usually not such a great idea. Password-based keys tend to have weak security, so they should be avoided whenever possible. Instead, it is better to use a truly random cryptographic key, e.g., a truly random 128-bit AES key, and find some way for the user to store it securely.

14.10. Alternatives to passwords

Finally, it is worth noting that there are many alternatives to passwords, for authenticating to a server. Some examples include:

- Two-factor authentication.
- One-time PINs (e.g., a single-use code sent via SMS to your phone, or a hardware device such as RSA SecurID).
- Public-key cryptography (e.g., SSH).
- Secure persistent cookies.

We most likely won't have time to discuss any of these further in this class, but they are worth knowing about, for situations where you need more security than passwords can provide.

14.11. Summary

The bottom line is: don't store passwords in the clear. Instead, sites should store passwords in hashed form, using a slow cryptographic hash function and a random salt. If the user's password is w, one can store

$$s, H(H(H(\cdots(H(w,s))\cdots)))$$

in the database, where s is a random salt chosen randomly for that user and H is a standard cryptographic hash function.

15. Case Studies

Case Studies

TODO: Under construction.

16. Bitcoin

16. Bitcoin

16.1. Problem Statement

Bitcoin is a digital cryptocurrency, which means it should have all the same properties as physical currency (e.g. the United States dollar). In our simplified model, a functioning currency should have the following properties:

- Each person has a bank account, in which they can store units of currency they own.
- Alice cannot impersonate Bob and perform actions as Bob.
- Any two people can engage in a *transaction*. Alice can send Bob n units of currency. This will cause Alice's bank account balance to decrease by n units, and Bob's bank account to increase by n units.
- If Alice has n units of currency in her account, she cannot spend any more than n units in any transaction.

In traditional physical currency, these properties are enforced by a trusted, centralized party such as a bank. Everyone trusts the bank to keep an accurate list of account holders with their appropriate account balances, and ensure that the identity of each user is correct before proceeding with a transaction. So, if Alice sends n units to Bob, both Alice and Bob trust that the bank will correctly decrease Alice's balance by n and increase Bob's balance by n. Everyone also trusts that the bank will not let Alice spend $n + 1$ units of currency if she only has n units in her account.

The goal of Bitcoin is to replicate these basic properties of a functioning currency system, but without any centralized party. Instead of relying on a trusted entity, Bitcoin uses cryptography to enforce the basic properties of currency.

16.2. Cryptographic Primitives

Bitcoin uses two cryptographic primitives that you have already seen in this class. Let's briefly review their definitions and relevant properties.

A *cryptographic hash* is a function H that maps arbitrary-length input x to a fixed-length output $H(x)$. The hash is collision-resistant, which means it is infeasible to find two different inputs that map to the same output. In math, it is infeasible to find $x \neq y$ such that $H(x) = H(y)$.

A *digital signature* is a cryptographic scheme that guarantees authenticity on a message. Alice generates a public verification key PK and a secret signing key SK. She broadcasts the public key to the world and keeps the secret key to herself. When Alice writes a message, she uses the secret key to generate a signature on her message and attaches the signature to the message. Anyone else can now use the public key to verify that the signature is valid, proving that the message was written by Alice and nobody tampered with it.

With these two cryptographic primitives in mind, we can now start designing Bitcoin.

16.3. Identities

Since there is no centralized party to keep track of everyone's accounts, we will need to assign a unique identity to everyone. We also need to prevent malicious users from pretending to be other users.

Every user of Bitcoin generates a public key and private key. Their identity is the public key. For example, Bob generates PK_B and SK_B and publishes PK_B to the world, so now his identity in Bitcoin is PK_B. When Bob is interacting with Bitcoin, he can prove that he is the user corresponding to PK_B by creating a message and signing it with SK_B. Then anybody can use PK_B to verify his signature and confirm that he is indeed the PK_B user. Because digital signatures are unforgeable, an attacker who doesn't know Bob's secret signing key will be unable to impersonate Bob, because the attacker cannot generate a signature that validates with PK_B.

16.4. Transactions

Without a centralized party to validate transactions, we will need a way to cryptographically verify that Alice actually wants to send n units of currency to Bob. Fortunately, this problem is essentially solved with our identity scheme above. If Alice wants to send n units of currency to Bob, she can create a message "PK_A sends n units of currency to PK_B" and sign it with her secret key. Note how she uses her public key PK_A as her identity and Bob's public key PK_B as his identity. Now anybody can verify the signature with Alice's public key to confirm that the user PK_A did intend to make this transaction. Bitcoin doesn't validate the recipient–if someone wanted to refuse a transaction, they could create another transaction to send the money back.

16.5. Balances

In our transaction scheme so far, nothing is stopping Alice from creating and signing a message "PK_A sends $100n$ units of currency to PK_B," even though she may only have n units of currency to spend. We need some way to keep track of each user's balances.

For now, assume that there is a *trusted ledger*. A ledger is a written record that everybody can view. It is append-only and immutable, which means you can only add new entries to the ledger, and you cannot change existing entries in the ledger. You can think of the ledger like a guest book: when you visit, you can add your own entry, and you can view existing entries, but you cannot (or should not) change other people's old entries. Later we will see how to build a decentralized ledger using cryptography.

Bitcoin does not explicitly record the balance of every user. Instead, every completed transaction (along with its signature) is recorded in the public ledger. Since everyone can view the ledger, anybody can identify an invalid transaction, such as Alice trying to spend more than she has. For example, suppose Bob starts with \$10 and everyone else starts with \$0. (We will discuss where Bob got the \$10 later.) Consider the following ledger:

- PK_B (Bob) sends PK_A (Alice) \$5. Message signed with SK_B.
- PK_B (Bob) sends PK_M (Mallory) \$2. Message signed with SK_B.
- PK_M (Mallory) sends PK_A (Alice) \$1. Message signed with SK_M.
- PK_A (Alice) sends PK_E (Eve) \$9. Message signed with SK_A.

Can you spot the invalid transaction? Although we don't have the balances of each user, the transaction ledger gives us enough information to deduce every user's balance at any given time. In this example, after the first three transactions, Bob has \$3, Mallory has \$1, and Alice has \$6. In the fourth transaction, Alice is trying to spend \$9 when she only has \$6, so we know it must be an invalid transaction. Because the ledger is trusted, it will reject this invalid transaction.

Thus, the idea is to have each block have a list of the transactions that show where the money being used in this transaction came from, which also means that blocks have to be sorted in order of creation. Now, our ledger looks as follows (again assuming that Bob starts with $10B$ and everyone else starts with $0B$:

- $TX_1 = PK_B$ (Bob) sends PK_A (Alice) $5B$, and the money came from the initial budget. TX_1 signed with SK_B
- $TX_2 = PK_A$ (Alice) sends PK_E (Eve) $5B$, and the money came from TX_1. TX_2 signed with SK_A

So, to check a transaction, we follow three steps:

1. Check that the signature on the transaction is verified using the PK of the sender
2. Check that the sender in this transaction was the receiver in some previous transaction
3. Check that the sender in this transaction has not spent the money in some previous transaction (aka they have enough money left over)

If we were checking TX_2, we first check that TX_2 was actually signed by Alice. Then, we check that Alice received some money in the past by checking the previous transactions. In TX_2, we see that Alice received the money from TX_1, and checking TX_1 verifies that Alice was the receiver. Next, we check that Alice has not spent the money earlier, so we scan the history of the blockchain and we don't see anywhere where the money from TX_1 was used. Finally, we check that Alice has 5 B by again checking TX_1 and seeing that she did receive 5 B from Bob. At this point, we have verified that TX_2 is a valid transaction, and we thus append it to the blockchain ledger.

At this point, we have created a functioning currency:

- Each person has a unique account, uniquely identified by public key.
- Users cannot impersonate other users, because each user can be validated by a secret signing key that only that user knows.
- Users can engage in a transaction by having the sender add their transaction to the ledger, with a signature on the transaction.
- Users cannot spend more than their current balance, because the trusted ledger is append-only, and everyone is able to calculate balances from the ledger.

The only remaining design element is creating a decentralized append-only ledger, which we will discuss next.

16.6. Hash chains

Recall that we need a public ledger that is append-only and immutable: everyone can add entries to the ledger, but nobody can modify or delete existing entries.

To build this ledger, we will start with a *hash chain*. Suppose we have five messages, $m_1, m_2, \ldots, m_5$ that we want to append to the ledger. The resulting hash chain would look like this:

Block 1	Block 2	Block 3	Block 4	Block 5
m_1	$m_2, H(\text{Block 1})$	$m_3, H(\text{Block 2})$	$m_4, H(\text{Block 3})$	$m_5, H(\text{Block 4})$

Note that each block contains the hash of the previous block, which in turn contains the hash of the previous block, etc. In other words, each time we append a new message in a new block, the hash of the previous block contains a digest of all the entries in the hash chain so far.

Another way to see this is to write out the hashes. For example:

$$\begin{aligned}\text{Block 4} &= m_4, H(\text{Block 3}) \\ &= m_4, H(m_3, H(\text{Block 2})) \\ &= m_4, H(m_3, H(m_2, H(\text{Block 1}))) \\ &= m_4, H(m_3, H(m_2, H(m_1)))\end{aligned}$$

Note that Block 4 contains a digest of all the messages so far, namely m_1, m_2, m_3, m_4.

16.7. Properties of Hash Chains

Assume that Alice is given the $H(\text{Block } i)$ from a trusted source, but she downloads blocks 1 through i from an untrusted source. Only using the $H(\text{Block } i)$, Alice can verify that the blocks she downloaded from the untrusted source are not compromised by recomputing the hashes of each block, checking that they match the hash in the next block, and so on, until the last block, which she checks against the hash she received from the trusted source. Let's walk through an example:

Say Alice received the $H(\text{Block } 4)$ from somewhere she trusts and then fetches the entire blockchain from a compromised server (so she downloads blocks 1 through 4). Can an attacker give Alice an incorrect chain, say with block 2 being incorrect, without her detecting it? No! Since we use cryptographic hashes, which are collision resistant, two different blocks cannot hash to the same value. Say that block 2 is incorrect and Alice instead received block $2'$, then $H(\text{Block } 2') \neq H(\text{Block } 2)$. Since block 3 includes the hash of block $2'$, block 3 will also be incorrect, so the third block that Alice received is block $3' \neq$ block 3. So, $H(\text{Block } 3') \neq H(\text{Block } 3)$. Then, since block 4 includes the hash of block $3'$, block 4 will also be incorrect, so the fourth block that Alice received is block $4' \neq$ block 4. So, $H(\text{Block } 4') \neq H(\text{Block } 4)$. Since Alice received H(block 4) from a trusted source, and it does not match up with $H(\text{Block}4')$, Alice is able to detect misbehavior. On the other hand, if the $H(\text{Block } 4')$ did match $H(\text{Block } 4)$, then the blockchain that Alice downloaded is correct, and we have no misbehavior.

So, perhaps the most important property in a hash chain is that if you get the hash of the latest block from a trusted source, then you can verify that all of the previous history is correct.

16.8. Consensus in Bitcoin

In Bitcoin, every participant in the network stores the entire blockchain (and thus all of its history) since we don't utilize a centralized server. When someone wants to create a new transaction, they broadcast that transaction to everyone, and each user on the network has to check the transaction. If the transaction is correct, they will append it to their local blockchain.

The issue is that some users might be malicious, meaning that they might not append certain transactions or might not check certain transactions correctly or might replay certain transactions or might allow invalid transactions. Bitcoin, however, assumes that the majority of users are honest.

Perhaps one of the biggest issues is forks, which are essentially different versions of the blockchain that exist at the same time. For example, say that Mallory bought a house from Bob for 500 B, and this transaction is appended to the ledger. Mallory can then try "go back in time" and start the blockchain from just before this transaction was added to it, and can start appending new transaction entries from there. If Mallory can get other users to accept this new forked chain, she can get her 500 B back!

This means that we need a way for all users to agree on the content of the blockchain: *consensus via proof of work*.

16.9. Consensus via Proof of Work

In Bitcoin, while every user locally stores the entire blockchain, not every user can add a block. This special privilege is reserved for certain users, known as *miners*, who can only add a block if they have a valid proof of work. A miner validates transactions before solving a *proof of work*, which, if completed before any other miner, allows the miner to append the block to the blockchain. The proof of work is a computational puzzle that takes the hash of the current block concatenated with a random number. This random number can be incremented so that the hash changes, until the proof of work is solved. The proof of work is considered solved when the resulting hash starts with N zero bits, where the value of N (e.g. 33) is determined by the Bitcoin algorithm.

Miners then broadcast blocks with their proof of work. All honest miners listen for such blocks, check the blocks for correctness, and *accept the longest correct chain*. If a miner appends a block with some incorrect transaction, the block is ignored. The key idea for consensus is that everyone will always prefer the longest correct chain. Thus, if multiple miners append blocks at the same time, consensus is gained by the longest

correct chain, and the rest of the "versions" are discarded. When two different miners at the same time solve a proof of work and append two different blocks, thus forking the network, the next miner that appends onto one of these chains invalidates the other chain.

Say for example that an honest miner M_1 stores the current local blockchain $b_1 \rightarrow b_2 \rightarrow b_3$, and hears about transaction T. M_1 checks T, then tries to mine (solve for the proof of work) for a new block b_4 to now include transaction T. However, if miner M_2 mines b_4 first, M_2 will broadcast $b_1 \rightarrow b_2 \rightarrow b_3 \rightarrow b_4$. M_1 checks b_4, accepts it, gives up mining block 4, then starts to mine for block 5. M_1 now has the blockchain $b_1 \rightarrow b_2 \rightarrow b_3 \rightarrow b_4$ stored locally and has started to mine b_5. However, if M_1 hears miner 3 broadcasts $b_1 \rightarrow b_2 \rightarrow b_3 \rightarrow b'_4 \rightarrow b'_5$, M_1 will discard the shorter blockchain ($b_1 \rightarrow b_2 \rightarrow b_3 \rightarrow b_4$) in favor of the longer one ($b_1 \rightarrow b_2 \rightarrow b_3 \rightarrow b'_4 \rightarrow b'_5$). By always accepting the longest blockchain, all the miners are ensured to have the same blockchain view.

Remember that Bitcoin assumes that more than half of the users are honest, meaning that more than half of the computing power is in the hands of honest miners, thus ensuring that honest miners will always have an advantage to mine the longest chain. Going back to the example about forks that prompted this discussion, if proof of consensus is implemented, Mallory cannot fork the blockchain since she does not have >50% of the computing power in the world. Since the longest chain is always taken as the accepted, Mallory's forked chain will be shorter unless she can mine new entries faster than the aggregate mining power of everyone else in the world.

Go forth and mine!

Web Security

Web Security

It would not be too much of a stretch to say that much of today's world is built upon the Internet. Many of the services that run on top of the Internet come with their own class of vulnerabilities and defenses to match. In particular, we will be focusing on web security, which covers a class of attacks that target web pages and web services.

17. SQL Injection

17. SQL Injection

17.1. Code Injection

SQL injection is a special case of a more broad category of attacks called code injections.

As an example, consider a calculator website that accepts user input and calls `eval` in Python in the server backend to perform the calculation. For example, if a user types 2+3 into the website, the server will run eval('2+3') and return the result to the user.

If the web server is not careful about checking user input, an attacker could provide a malicious input like

2+3"); os.system("rm -rf /

When the web server plugs this into the `eval` function, the result looks like

eval("2+3"); os.system("rm .")

If interpreted as code, this statement causes the web server to delete all its files!

The general idea behind these attacks is that a web server uses user input as part of the code it runs. If the input is not properly checked, an attacker could create a special input that causes unintended code to run on the server.

17.2. SQL Injection Example

Many modern web servers use SQL databases to store information such as user logins or uploaded files. These servers often allow users to interact with the database through HTTP requests.

For example, consider a website that stores a SQL table of course evaluations named `evals`:

id	course	rating
1	cs61a	4.5
2	cs61b	4.4
3	cs161	5.0

A user can make an HTTP GET request for a course rating through a URL:

http://www.berkeley.edu/evals?course=cs61a

To process this request, the server performs a SQL query to look up the rating corresponding to the course the user requested:

SELECT rating FROM evals WHERE course = 'cs61a'

Just like the code injection example, if the server does not properly check user input, an attacker could create a special input that allows arbitrary SQL code to be run. Consider the following malicious input:

garbage'; SELECT password FROM passwords WHERE username = 'admin

When the web server plugs this into the SQL query, the resulting query looks like

SELECT rating FROM evals WHERE course = 'garbage'; SELECT * FROM passwords WHERE username = 'admin'

If interpreted as code, this causes the query to return the password for the `admin` user!

17.3. SQL Injection Strategies

Writing a malicious input that creates a syntactically valid SQL query can be tricky. Let's break down each part of the malicious input from the previous example:

- `garbage` is a garbage input to the intended query so that it doesn't return anything.
- `'` closes the opening quote from the intended query. Without this closing quote, the rest of our query would be treated as a string, not SQL code.
- `;` ends the intended SQL query and lets us start a new SQL query.
- `SELECT password FROM passwords WHERE username = 'admin` is the malicious SQL query we want to execute. Note that we didn't add a closing quote to `'admin`, because the intended SQL query will automatically add a closing quote at the end of our input.

Consider another vulnerable SQL query. This time, we have a `users` table that contains the `username` and `password` of every user.

When the web server receives a login request, it creates a SQL query by plugging in the username and password from the request. For example, if you make a login request with username `alice` and password `password123`, the resulting SQL query would be

SELECT username FROM users WHERE username = 'alice' AND password = 'password123'

If the query returns more than 0 rows, the server registers a successful login.

Suppose we want to login to the server, but we don't have an account, and we don't know anyone's username. How might we achieve this using SQL injection?

First, in the username field, we should add a dummy username and a quote to end the opening quote from the original query:

SELECT username FROM users WHERE username = 'alice'' AND password = 'password123'

Next, we need to add some SQL syntax so that this query returns more than 0 rows (since we don't know if `alice` is a valid username). One trick for forcing a SQL query to always return something is to add some logic that always evaluates to true, such as `OR 1=1`:

SELECT username FROM users WHERE username = 'alice' OR 1=1' AND password = '_____'

Next, we have to add some SQL so that the rest of the query doesn't throw a syntax error. One way of doing this is to add a semicolon (ending the previous query) and write a dummy query that matches the remaining SQL:

SELECT username FROM users WHERE username = 'alice' OR 1=1; SELECT username FROM users WHERE username = 'alice' AND password = '_____'

The second query might not return anything, but the first query will return a nonzero number of entries, which lets us perform a login. The last step is to add some garbage as the password:

SELECT username FROM users WHERE username = 'alice' OR 1=1; SELECT username FROM users WHERE username = 'alice' AND password = 'garbage'

Thus, our malicious username and password should be

username = alice' OR 1=1; SELECT username FROM users WHERE username = 'alice password = garbage

Another trick to make SQL injection easier is the `--` syntax, which starts a comment in SQL. This tells SQL to ignore the rest of the query as a comment.

In our previous example, we can instead start a comment to ignore parts of the query we don't want to execute:

SELECT username FROM users WHERE username = 'alice' OR 1=1–' AND password = 'garbage'

Thus, another malicious username and password is

username = alice' OR 1=1– password = garbage

Further reading: SQL Injection Attacks by Example

17.4. Defense: Escape Inputs

One way of defending against SQL injection is to escape any potential input that could be used in an attack. Escaping a character means that you tell SQL to treat this character as part of the string, not actual SQL syntax.

For example, the quote `"` is used to denote the end of a string in SQL. However, the escaped quote `\"` is treated as a literal quote character in SQL, and it does not cause the current string to end.

By properly replacing characters with their escaped version, malicious inputs such as the ones we've been creating will be treated as strings, and the SQL parser won't try to run them as actual SQL commands.

For example, in the previous exploit, if the server replaces all instances of the quote `"` and the dash `-` with escaped versions, the SQL parser will see

SELECT username FROM users WHERE username = 'alice' OR 1=1--' AND password = 'garbage'

The escaped quote won't cause the `username` string to end, and the escaped dashes won't cause a comment to be created. The parser will try to look up someone with a username `alice" OR 1=1--` and find nothing.

However, we have to be careful with escaping. If an attacker inputs a backslash followed by a quote `\"`, the escaper might "escape the escape" and give the input `\\"` to the SQL parser. The parser will treat the two backslashes `\\` as an escaped backslash, and the quote won't be escaped!

The key takeaway here is that building a good escaper can be tricky, and there are many edge cases to consider. There is almost no circumstance in which you should try to build an escaper yourself; secure SQL escapers exist in SQL libraries for almost every programming language. However, if you are running SQL statements with raw user input, escapers are often an ineffective solution, because you need to ensure that every call is properly escaped. A far more robust solution is to use parameterized SQL.

17.5. Defense: Parameterized SQL/Prepared Statements

A better defense against SQL injection is to use parameterized SQL or prepared statements. This type of SQL compiles the query first, and then plugs in user input after the query has already been interpreted by the SQL parser. Because the user input is added after the query is compiled and interpreted, there is no way for any attacker input to be treated as SQL code. Parameterized SQL prevents all SQL injection attacks, so it is the best defense against SQL injection!

In most SQL libraries, parameterized SQL and unsafe, non-paramaterized SQL are provided as two different API functions. You can ensure that you've eliminated *all* potential SQL vulnerabilities in your code by searching for every database query and replacing each API call with a call to the parameterized SQL API function.

The biggest problem with parameterized SQL is compatibility. SQL is a (mostly) generic language, so SQL written for MySQL can run on Postgres or commercial databases. Parameterized SQL requires support from the underlying database (since the processing itself happens on the database side), and there is no common standard for expressing parameterized SQL. Most SQL libraries will handle the translation for you, but switching to prepared statements may make it harder to switch between databases.

In practice, most modern SQL libraries support parameterized SQL and prepared statements. If the library you are using does not support parameterized SQL, it is probably best to switch to a different SQL library.

Further reading: OWASP Cheat Sheet on SQL Injection

18. Introduction to the Web

18. Introduction to the Web

18.1. URLs

Every resource (webpage, image, PDF, etc.) on the web is identified by a URL (Uniform Resource Locator). URLs are designed to describe exactly where to find a piece of information on the Internet. A basic URL consists of three mandatory parts:

http://www.example.com/index.html

The first mandatory part is the *protocol*, located before in the URL. In the example above, the protocol is http. The protocol tells your browser how to retrieve the resource. In this class, the only two protocols you need to know are HTTP, which we will cover in the next section, and HTTPS, which is a secure version of HTTP using TLS (refer to the networking unit for more details). Other protocols include `git+ssh://`, which fetches a git archive over an encrypted tunnel using `ssh`, or `ftp://`, which uses the old FTP (File Transfer Protocol) to fetch data.

The second mandatory part is the *location*, located after but before the next forward slash in the URL. In the example above, the location is www.example.com. This tells your browser which web server to contact to retrieve the resource.

Optionally, the location may contain an optional *username*, which is followed by an `@` character if present. For example, evanbot@www.example.com is a location with a username `evanbot`. All locations must include a computer identifier. This is usually a domain name such as www.example.com. Sometimes the location will also include a port number, such as www.example.com:81, to distinguish between different applications running on the same web server. We will discuss ports a bit more when we talk about TCP during the networking section.

The third mandatory part is the *path*, located after the first single forward slash in the URL. In the example above, the path is /index.html. The path tells your browser which resource on the web server to request. The web server uses the path to determine which page or resource should be returned to you.

One way to think about paths is to imagine a filesystem on the web server you're contacting. The web server can use the path as a filepath to locate a specific page or resource. The path must at least consist of `/`, which is known as the "root"[1] of the filesystem for the remote web site.

Optionally, there can be a `?` character after the path. This indicates that you are supplying additional arguments in the URL for the web server to process. After the `?` character, you can supply an optional set of *parameters* separated by `&` characters. Each parameter is usually encoded as a key-value pair in the format `key=value`. Your browser sends all this information to the web server when fetching a URL. See the next section for more details on URL parameters.

Finally, there can be an optional *anchor* after the arguments, which starts with a `#` character. The anchor text is not sent to the server, but is available to the web page as it runs in the browser.

The anchor is often used to tell your browser to scroll to a certain part of the webpage when loading it. For example, try loading https://en.wikipedia.org/wiki/Dwinelle_Hall#Floor_plan and https:

[1] It is called the root because the filesystem can be treated as a tree and this is where the tree starts.

//en.wikipedia.org/wiki/Dwinelle_Hall#Construction and note that your browser skips to the section of the article specified in the anchor.

In summary, a URL with all elements present may look like this:

http://evanbot@www.cs161.org:161/whoami?k1=v1&k2=v2#anchor

where http is the protocol, evanbot is the username, www.cs161.org is the computer location (domain), 161 is the port, /whoami is the path, k1=v1&k2=v2 are the URL arguments, and anchor is the anchor.

Further reading: What is a URL?

18.2. HTTP

The protocol that powers the World Wide Web is the Hypertext Transfer Protocol, abbreviated as HTTP. It is the language that clients use to communicate with servers in order to fetch resources and issue other requests. While we will not be able to provide you with a full overview of HTTP, this section is meant to get you familiar with several aspects of the protocol that are important to understanding web security.

18.3. HTTP: The Request-Response Model

Fundamentally, HTTP follows a request-response model, where clients (such as browsers) must actively start a connection to the server and issue a request, which the server then responds to. This request can be something like "Send me a webpage" or "Change the password for my user account to `foobar`." In the first example, the server might respond with the contents of the web page, and in the second example, the response might be something as simple as "Okay, I've changed your password." The exact structure of these requests will be covered in further detail in the next couple sections.

The original version of HTTP, HTTP 1.1, is a text-based protocol, where each HTTP request and response contains a *header* with some metadata about the request or response and a *payload* with the actual contents of the request or response. HTTP2, a more recent version of HTTP, is a binary-encoded protocol for efficiency, but the same concepts apply.

For all requests, the server generates and sends a response. The response includes a series of headers and, in the payload, the body of the data requested.

18.4. HTTP: Structure of a Request

Below is a very simple HTTP request.

```
GET / HTTP/1.1
Host: squigler.com
Dnt: 1
```

The first line of the request contains the method of the request (`GET`), the path of the request (`/`), and the protocol version (`HTTP/1.1`). This is an example of a GET request. Each line after the first line is a request header. In this example, there are two headers, the DNT header and the Host header. There are many HTTP headers defined in the HTTP spec which are used to convey various pieces of information, but we will only be covering a couple of them through this chapter.

Here is another HTTP request:

```
POST /login HTTP/1.1
Host: squigler.com
Content-Length: 40
Content-Type: application/x-url-formencoded
Dnt: 1

username=alice@foo.com&password=12345678
```

Here, we have a couple more headers and a different request type: the POST request.

18.5. HTTP: GET vs. POST

While there are quite a few methods for requests, the two types that we will focus on for this course are GET requests and POST requests. GET requests are generally intended for "getting" information from the server. POST requests are intended for sending information to the server that somehow modifies its internal state, such as adding a comment in a forum or changing your password.

In the original HTTP model, GET requests are not supposed to change any server state. However, modern web applications often change server state in response to GET requests in query parameters.

Of note, only POST requests can contain a body in addition to request headers. Notice that the body of the second example request contains the username and password that the user `alice` is using to log in. While GET requests cannot have a body, it can still pass query parameters via the URL itself. Such a request might look something like this:

```
GET /posts?search=security&sortby=popularity
Host: squigler.com
Dnt: 1
```

In this case, there are two query parameters, `search` and `sortby`, which have values of `security` and `popularity`, respectively.

18.6. Elements of a Webpage

The HTTP protocol is designed to return arbitrary files. The response header usually specifies a media type that tells the browser how to interpret the data in the response body.

Although the web can be used to return files of any type, much of the web is built in three languages that provide functionality useful in web applications.

A modern web page can be thought of as a distributed application: there is a component running on the web server and a component running in the web browser. First, the browser makes an HTTP request to a web server. The web server performs some server-side computation and generates and sends an HTTP response. Then, the browser performs some browser-side computation on the HTTP response and displays the result to the user.

18.7. Elements of a Webpage: HTML

HTML (Hypertext Markup Language) lets us create structured documents with paragraphs, links, fillable forms, and embedded images, among other features. You are not expected to know HTML syntax for this course, but some basics are useful for some of the attacks we will cover.

Here are some examples of what HTML can do:

- Create a link to Google: `<a href="http://google.com">Click me</a>`
- Embed a picture in the webpage: `<img src="http://cs161.org/picture.png">`
- Include JavaScript in the webpage: `<script>alert(1)</script>`
- Embed the CS161 webpage in the webpage: `<iframe src="http://cs161.org"></iframe>`

Frames pose a security risk, since the outer page is now including an inner page that may be from a different, possibly malicious source. To protect against this, modern browsers enforce frame isolation, which means the outer page cannot change the contents of the inner page, and the inner page cannot change the contents of the outer page.

18.8. Elements of a Webpage: CSS

CSS (Cascading Style Sheets) lets us modify the appearance of an HTML page by using different fonts, colors, and spacing, among other features. You are not expected to know CSS syntax for this course, but you should know that CSS is as powerful as JavaScript when used maliciously. If an attacker can force a victim to load some malicious CSS, this is functionally equivalent to the attacker forcing the victim to load malicious JavaScript.

18.9. Elements of a Webpage: JavaScript

JavaScript is a programming language that runs in your browser. It is a very powerful language–in general, you can assume JavaScript can arbitrarily modify any HTML or CSS on a webpage. Webpages can include JavaScript in their HTML to allow for dynamic features such as interactive buttons. Almost all modern webpages use JavaScript.

When a browser receives an HTML document, it first converts the HTML into an internal form called the DOM (Document Object Model). The JavaScript is then applied on the DOM to modify how the page is displayed to the user. The browser then renders the DOM to display the result to the user.

Because JavaScript is so powerful, modern web browsers run JavaScript in a sandbox so that any JavaScript code loaded from a webpage cannot access sensitive data on your computer or even data on other webpages.

Most exploits targeting the web browser itself require JavaScript, either because the vulnerability lies in the browser's JavaScript engine, or because JavaScript is used to shape the memory layout of the program for improving the success rate of an attack.

Almost all web browsers implement JavaScript as a Just In Time compiler, dynamically converting JavaScript into machine code[2]. Many modern desktop applications (notably Slack's desktop client) are actually written in the Electron framework, which is effectively a cut down web browser running JavaScript.

[2]Trivia: Running JavaScript fast is considered so important that ARM recently introduced a dedicated instruction, FJCVTZS (Floating-point Javascript Convert to Signed fixed-point, rounding toward Zero), specifically to handle how JavaScript's math operates.

19. Same-Origin Policy

19. Same-Origin Policy

Browsing multiple webpages poses a security risk. For example, if you have a malicious website (`www.evil.com`) and Gmail (`www.gmail.com`) open, you don't want the malicious website to be able to access any sensitive emails or send malicious emails with your identity.

Modern web browsers defend against these attacks by enforcing the same-origin policy, which isolates every webpage in your browser, except for when two webpages have the same origin.

19.1. Origins

The origin of a webpage is determined by its protocol, domain name, and port. For example, the following URL has protocol `http`, domain name `www.example.com`, and port `80`.

http://www.example.com/index.html

To check if two webpages have the same origin, the same-origin policy performs string matching on the protocol, domain, and port. Two websites have the same origin if their protocols, domains, and ports all exactly match.

Some examples of the same origin policy:

- `http://wikipedia.org/a/` and `http://wikipedia.org/b/` have the same origin. The protocol (`http`), domain (`wikipedia.org`), and port (none), all match. Note that the paths are not checked in the same-origin policy.
- `http://wikipedia.org` and `http://www.wikipedia.org` do not have the same origin, because the domains (`wikipedia.org` and `www.wikipedia.org`) are different.
- `http://wikipedia.org` and `https://wikipedia.org` do not have the same origin, because the protocols (`http` and `https`) are different.
- `http://wikipedia.org:81` and `http://wikipedia.org:82` do not have the same origin, because the ports (`81` and `82`) are different.

If a port is not specified, the port defaults to 80 for http and 443 for https. This means `http://wikipedia.org` has the same origin as `http://wikipedia.org:80`, but it does not have the same origin as `http://wikipedia.org:81`.

19.2. Exceptions

In general, the origin of a webpage is defined by its URL. However, there are a few exceptions to this rule:

- JavaScript runs with the origin of the page that loads it. For example, if you include `<script src="http://google.com/tracking.js"></script>` on `http://cs161.org`, the script has the origin of `http://cs161.org`.
- Images have the origin of the page that it comes from. For example, if you include `<img src="http://google.com/logo.jpg">` on `http://cs161.org`, the image has the origin of

`http://google.com`. The page that loads the image (`http://cs161.org`) only knows about the image's dimensions when loading it.

- Frames have the origin of the URL where the frame is retrieved from, not the origin of the website that loads it. For example, if you include `<iframe src="http://google.com"></iframe>` on `http://cs161.org`, the frame has the origin of `http://google.com`.

JavaScript has a special function, `postMessage`, that allows webpages from different origins to communicate with each other. However, this function only allows very limited functionality.

Further reading: Same-origin policy

20. Cookies and Session Management

20. Cookies and Session Management

HTTP is a stateless protocol, which means each request and response is independent from all other requests and responses. However, many features on the web require maintaining some form of state. For example, when you log into your email account, you can stay logged in across many requests and responses. If you enable dark mode on a website and make subsequent requests to the website, you want the pages returned to have a dark background. If you're browsing an online shopping website, you want the items in your cart to be saved across many requests and responses. Browser and servers store HTTP cookies to support these features.

At a high level, you can think of cookies as pieces of data stored in your browser. When you make a request to enable dark mode or add an item to your shopping cart, the server sends a response with a `Set-Cookie` header, which tells your browser to store a new cookie. These cookies encode state that should persist across multiple requests and responses, such as your dark mode preference or a list of items in your shopping cart. In future requests, your browser will automatically attach the relevant cookies to a request and send it to the web server. The additional information in these cookies helps the web server customize its response.

20.1. Cookie Attributes

Every cookie is a name-value pair. For example, a cookie `darkmode=true` has name `darkmode` and value `true`.

For security and functionality reasons, we don't want the browser to send every cookie in every request. A user might want to enable dark mode on one website but not on another website, so we need a way to only send certain cookies to certain URLs. Also, as we'll see later, cookies may contain sensitive login information, so sending all cookies in all requests poses a security risk. These additional cookie attributes help the browser determine which cookies should be attached to each request.

- The `Domain` and `Path` attributes tell the browser which URLs to send the cookie to. See the next section for more details.
- The `Secure` attribute tells the browser to only send the cookie over a secure HTTPS connection.
- The `HttpOnly` attribute prevents JavaScript from accessing and modifying the cookie.
- The `expires` field tells the browser when to stop remembering the cookie.

20.2. Cookie Policy: Domain and Path

The browser sends a cookie to a given URL if the cookie's `Domain` attribute is a domain-suffix of the URL domain, and the cookie's `Path` attribute is a prefix of the URL path. In other words, the URL domain should end in the cookie's `Domain` attribute, and the URL path should begin with the cookie's `Path` attribute.

For example, a cookie with Domain=example.com and Path=/some/path will be included on a request to http://foo.example.com/some/path/index.html, because the URL domain ends in the cookie domain, and the URL path begins with the cookie path.

Note that cookie policy uses a different set of rules than the same origin policy. This has caused problems in the past.

20.3. Cookie Policy: Setting Domain and Path

For security reasons, we don't want a malicious website `evil.com` to be able to set a cookie with domain `bank.com`, since this would allow an attacker to affect the functionality of the legitimate bank website. To prevent this, the cookie policy specifies that when a server sets a cookie, the cookie's domain must be a URL suffix of the server's URL. In other words, for the cookie to be set, the server's URL must end in the cookie's `Domain` attribute. Otherwise, the browser will reject the cookie.

For example, a webpage with domain `eecs.berkeley.edu` can set a cookie with domain `eecs.berkeley.edu` or `berkeley.edu`, since the webpage domain ends in both of these domains.

This policy has one exception: cookies cannot have domains set to a top-level domain, such as `.edu` or `.com`, since these are too broad and pose a security risk. If `evil.com` could set cookies with domain `.com`, the attacker would have the ability to affect all `.com` websites, since this cookie would be sent to all `.com` websites. The web browser maintains a list of top-level domains, which includes two-level TLDs like `.co.uk`.

The cookie policy allows a server to set the `Path` attribute without any restrictions.[1]

Further reading: Cookies

20.4. Session Management

Cookies are often used to keep users logged in to a website over many requests and responses. When a user sends a login request with a valid username and password, the server will generate a new session token and send it to the user as a cookie. In future requests, the browser will attach the session token cookie and send it to the server. The server maintains a mapping of session tokens to users, so when it receives a request with a session token cookie, it can look up the corresponding user and customize its response accordingly.

Secure session tokens should be random and unpredictable, so an attacker cannot guess someone else's session token and gain access to their account. Many servers also set the `HttpOnly` and `Secure` flags on session tokens to protect them from being accessed by XSS vulnerabilities or network attackers, respectively.

It is easy to confuse session tokens and cookies. Session tokens are the values that the browser sends to the server to associate the request with a logged-in user. Cookies are how the browser stores and sends session tokens to the server. Cookies can also be used to save other state, as discussed earlier. In other words, session tokens are a special type of cookie that keep users logged in over many requests and responses.

[1] The lack of restriction on the `Path` attribute has caused problems in the past, as cookies are presented to the server and JavaScript as an unordered set of name/value pairs, but is stored internally as name/path/value tuples, so if two cookies with the same name and host but different path are present, both will be presented to the server in unspecified order.

21. Cross-Site Request Forgery (CSRF)

21. Cross-Site Request Forgery (CSRF)

21.1. CSRF Attacks

Using cookies and session tokens to keep a user logged in has some associated security risks. In a cross-site request forgery (CSRF) attack, the attacker forces the victim to make an unintended request. The victim's browser will automatically attach the session token cookie to the unintended request, and the server will accept the request as coming from the victim.

For example, suppose a website has an endpoint `http://example.com/logout`. To log out, a user makes a GET request to this URL with the appropriate session token attached, and the server checks the session token and performs the logout. If an attacker can trick a victim into clicking this link, the victim will be logged out of the website without their knowledge.

CSRF attacks can also be executed on URLs with more malicious actions. For example, a GET request to `https://bank.com/transfer?amount=100&recipient=mallory` with a valid session token might send $100 to Mallory. An attacker could send an email to the victim with the following HTML snippet:

This will cause the browser to try and fetch an image from the malicious URL by making a GET request. Because the browser automatically attaches the session token to the request, this causes the victim to unknowingly send $100 to Mallory.

It is usually bad practice to have HTTP GET endpoints that can change server state, so this type of CSRF attack is less common in practice. However, CSRF attacks are still possible over HTTP POST requests. HTML forms are a common example of a web feature that generates HTTP POST requests. The user fills in the form fields, and when they click the Submit button, the browser generates a POST request with the filled-out form fields. Consider the following HTML snippet on an attacker's webpage:

```
<form name=evilform action=https://bank.com/transfer>
  <input name=amount value=100>
  <input name=recipient value=mallory>
</form>
<script>document.evilform.submit();</script>
```

When the victim visits the attacker's website, this HTML snippet will cause the victim's browser to make a POST request to `https://bank.com/transfer` with form input values that transfer $100 to Mallory. Like before, the victim's browser automatically attaches the session token to the request, so the server accepts this POST request as if it was from the victim.

21.2. Defense: CSRF Token

A good defense against CSRF attacks is to include a CSRF token on webpages. When a legitimate user loads a webpage from the server with a form, the server will randomly generate a CSRF token and include it as an extra field in the form. (In practice, this field often has a hidden attribute set so that it's only visible in the HTML, so users don't see random strings every time they submit a form.) When the user submits the form,

the form will include the CSRF token, and the server will check that the CSRF token is valid. If the CSRF token is invalid or missing, the server will reject the request.

To implement CSRF tokens, the server needs to generate a new CSRF token every time a user requests a form. CSRF tokens should be random and unpredictable so an attacker cannot guess the CSRF token. The server also needs to maintain a mapping of CSRF tokens to session tokens, so it can validate that a request with a session token has the correct corresponding CSRF token. This may require the server to store a large amount of state if it expects heavy traffic.

If an attacker tries the attack in the previous section, the malicious form they create on their website will no longer contain a valid CSRF token. The attacker could try querying the server for a CSRF token, but it would not properly map to the victim's session token, because the victim never requested the form legitimately.

21.3. Defense: Referer Validation

Another way to defend against CSRF tokens is to check the Referer[1] field in the HTTP header. When a browser issues an HTTP request, it includes a Referer header which indicates which URL the request was made from. For example, if a user fills out a form from a legitimate bank website, the Referer header will be set to `bank.com`, but if the user visits the attacker's website and the attacker fills out a form and submits it, the Referer header will be set to `evil.com`. The server can check the Referer header on each request and reject any requests that have untrusted or suspicious Referer headers.

Referer validation is a good defense if it is included on every request, but it poses some problems if someone submits a request with the Referer header left blank. If a server accepts requests with blank Referer headers, it may be vulnerable to CSRF attacks, but if a server rejects requests with blank Referer headers, it may reduce functionality for some users.

In practice, Referer headers may be removed by the browser, the operating system, or a network monitoring system for privacy issues. For example, if you click on a link to visit a website from a Google search, the website can know what Google search you made to visit its website from the Referer header. Some modern browsers also have options that let users disable sending the Referer header on all requests. Because not all requests are guaranteed to have a Referer header, it is usually only used as a defense-in-depth strategy in addition to CSRF tokens, instead of as the only defense against CSRF attacks.

Further reading: OWASP Cheat Sheet on CSRF

[1] Yes, the "Referer" field represents a roughly three decade old misspelling of referrer. This is a silly example of how "legacy", that is old design decisions, can impact things decades later because it can be very hard to change things

22. Cross-Site Scripting (XSS)

22. Cross-Site Scripting (XSS)

XSS is a class of attacks where an attacker injects malicious JavaScript onto a webpage. When a victim user loads the webpage, the user's browser will run the malicious JavaScript.

XSS attacks are powerful because they subvert the same-origin policy. Normally, an attacker can only run JavaScript on websites they control (such as `https://evil.com`), so their JavaScript cannot affect websites with origins different from `https://evil.com`. However, if the attacker can inject JavaScript into `https://google.com`, then when a user loads `https://google.com`, their browser will run the attacker's JavaScript with the origin of `https://google.com`.

XSS attacks allow malicious JavaScript to run in the user's browser with the same origin as a legitimate website. This allows the attacker to perform any action the user can perform at `https://google.com` or steal any user secrets associated with Google and send them back to the attacker.

There are two main categories of XSS attacks: stored XSS and reflected XSS.

22.1. Stored XSS

In a stored XSS attack, the attacker finds a way to persistently store malicious JavaScript on the web server. When the victim loads the webpage, the web server will load this malicious JavaScript and display it to the user.

A classic example of stored XSS is a Facebook post. When a user makes a Facebook post, the contents of the post are stored on Facebook's servers, so that other users can load their friends' posts. If Facebook doesn't properly check user inputs, an attacker could make a post that says

<script>alert("XSS attack!")</script>

This post is now stored in Facebook's servers. If another user loads the attacker's posts, they will receive an HTML page with this script on it, and the browser will run the script and trigger a pop-up that says `XSS attack!`

22.2. Reflected XSS

In a reflected XSS attack, the attacker finds a vulnerable webpage where the server receives user input in an HTTP request and displays the user input in the response.

A classic example of reflected XSS is a Google search. When you make an HTTP GET request for a Google search, such as `https://www.google.com/search?&q=cs161`, the returned webpage with search results will include something like

You searched for: cs161

If Google does not properly check user input, an attacker could create a malicious URL `https://www.google.com/search?&q=<attack!")</script>`. When the victim loads this URL, Google will return

You searched for: <script>alert("XSS attack!")</script>

The victim's browser will run the script and trigger a pop-up that says `XSS attack!`

22.3. Defense: Sanitize Input

A good defense against XSS is checking for malicious input that might cause JavaScript to run, such as `<script>` tags. However, it is very difficult to write a good detector that catches all XSS attacks. For example, the following input causes JavaScript to run without ever using `<script>` tags:

Just like SQL input escaping, sanitizing potentially dangerous input can be very tricky. For example, consider an escaper that searches for all instances of `<script>` and `</script>` and removes them. An attacker could provide this malicious input:

<scr<script>ipt>alert("XSS attack!")</scr<script>ipt>

After the escaper removes the two `<script>` tags it sees, the result is `<script>alert("XSS attack!")</script>`, and the attacker can still execute JavaScript!

Another way to escape input is to replace potentially dangerous characters with their HTML encoding. For example, the less than (`<`) and greater than (`>`) signs are encoded as `&lt;` and `&gt;`, respectively. These encodings cause less than and greater than signs to display on the webpage, without being interpreted as HTML.

Fortunately, there is a standardized set of sanitizations that is known to be robust.

22.4. Defense: Content Security Policy

Another XSS defense is using a content security policy (CSP) that specifies a list of allowed domains where scripts can be loaded from. For example, `cs161.org` might allow scripts that are loaded from `*.cs161.org` or `*.google.com` and disallow all other scripts, including any inline scripts that are injected by the attacker.

CSPs are defined by a web server and enforced by a browser. In the HTTP response, the server attaches a `Content-Security-Policy` header, and the browser checks any scripts against the header.

If you enable CSP, you can no longer run *any* scripts that are embedded directly in the HTML document. You can only load external scripts specified by the `script` tag and an external URL. These scripts can only be fetched from the sites specified in the CSP. This prevents an attacker from directly injecting scripts into an HTML document or modifying the HTML document to fetch scripts from the attacker's domain.

Further reading: OWASP Cheat Sheet on XSS

23. UI Attacks

23. Clickjacking/User Interface (UI) Attacks

23.1. Clickjacking Attacks

Many of the web attacks we've seen involve forcing a victim to click on an attacker-generated link (reflected XSS), or forcing a victim to visit an attacker-controlled website (CSRF). How might an attacker achieve this?

UI attacks (or *clickjacking attacks*) are a category of attacks that try to fool a victim into inadvertently clicking on an attacker-supplied input. The end goal of these attacks is to "steal" a click from the user, so that the user loads something controlled by the attacker (possibly for a further attack). Many UI attacks rely on clever visual tricks to deceive the user.

Download buttons are a classic example of clickjacking. When you visit a website to download a file, you might see many different download buttons with different shapes and colors. One of these is the true download button, and the others are malicious download buttons that actually take you to attacker-controlled websites or perform other malicious actions in your browser. An unwitting user might click on the wrong download button and be sent to the attacker website. The malicious download buttons could be added to the website through a different web exploit (e.g. stored XSS) or as a paid advertisement.

Depending on how much control the attacker has over the page, more sophisticated clickjacking attacks are possible:

- The attacker could manipulate an HTML form so that the user sees a payment of $5, but the underlying form will actually submit a payment of $50.
- The attacker could draw a fake cursor on the page. The user sees the fake cursor over a legitimate button and clicks, but their real cursor has actually clicked on a malicious link.
- The attacker could draw an entire browser on the page. The user sees an address bar and clicks, but they have actually clicked on a fake address bar generated by the attacker (with a malicious link behind the address bar).

23.2. Clickjacking Defenses

There are many ways to defend against clickjacking attacks. The general idea is to force the user to make sure that they're clicking on what they intended to click.

Confirmation pop-ups: If the user clicks on a link or button that will perform some potentially dangerous activity (e.g. opening a website, executing Javascript, downloading a file), display a pop-up asking the user to confirm that this is their intended action. However, users might still click on the pop-up without reading it, especially if they're too frequent. Remember to consider human factors!

UI randomization: Randomize the location of certain elements on a website. For example, a submit button could alternately be located at the left side of the screen and the right side of the screen. This makes it harder for attackers to draw a fake submit button over the real submit button, because they won't know where it's located. However, webpages that look different every time could pose usability problems.

Direct the user's attention to their click: This can be done by freezing the rest of the screen besides the area directly around the user's cursor, or by highlighting the user's cursor. This will make the user less

likely to be fooled by a fake cursor and force them to focus on where their real cursor is pointing. The user's clicks can also be invalidated if the user tries to click outside of a relevant portion of the screen.

Delay the click: Force the user to hover over the desired button for some amount of time before allowing the user to click the button. This forces the user to spend some time looking at where they're clicking before they actually perform the click.

24. CAPTCHAs

24. CAPTCHAs

24.1. Using CAPTCHAs

Consider the following scenario: you've created a website that allows users to upload a picture. Your server will scan the picture for text using a compute-intensive algorithm and return the text to the user. An adversary wants to mount a denial-of-service (DoS) attack on your website by uploading lots of bogus images, forcing your server to run the expensive algorithm on all the bogus images.

Consider another scenario: Your website has a login page. An adversary wants to steal a legitimate user's account, so they try to brute-force the user's password by submitting login requests with every possible password.

Generally, when we're building websites, we'd like to build websites for people: we don't want robots. CAPTCHAs are a test that ask the fundamental question: *Is this a human?* Consequently, when we design CAPTCHAs, we want to choose problems that are easy for humans, but difficult for computers.

CAPTCHAs are primarily focused on machine vision problems, which are traditionally difficult for computers to solve. Historically, CAPTCHAs consist of a series of distorted letters or words. There are a wide variety of CAPTCHAs: some with color, some with low contrasts, some with merged-together letters, etc. A more recent example you may be familiar with is Google's reCAPTCHA algorithm, which shows you some images and asks you to identify the objects in the pictures (e.g. "Select all images with boats.")

24.2. Issues with CAPTCHAs

There's an inherent arms race present here: as solving algorithms get better, our defense deteriorates. The reason why CAPTCHAs have gotten so much harder over the last decade is because individuals have spent time creating much better solving algorithms - and we're reaching a point where it's becoming more and more difficult for humans to solve CAPTCHAs quickly.

Of course, those implementing CAPTCHAs often miss the original motivation behind their development. The original CAPTCHA paper included the subtitle "How Lazy Cryptographers do AI" as the intent was to force attackers to solve harder problems in machine vision. Now modern CAPTCHAs such as Google ReCAPTCHA are focused on getting humans to provide training data for AI systems which means the CAPTCHAs are inherently self defeating for those deploying the CAPTCHA.

In some cases, it's necessary to provide an alternative, accessible CAPTCHA method, such as an audio-based spoken phrase that a human is required to transcribe. In this case, we've unintentionally opened up a new attack vector: attackers may now target the audio-based CAPTCHA, which may be easier to solve than the traditional image-based CAPTCHA.

If you search "crack CAPTCHA" on Google, you'll likely find many CAPTCHA solving services for as low as $0.10 cents per CAPTCHA. These services use humans to do the actual work. These days, a CAPTCHA no longer asks the question of "Is this a human or a bot?" Instead, it says "Is this a human, or a bot willing to spend a fraction of a penny?"

The takeaway: if something is worth $0.10 or more to an attacker, CAPTCHAs do not work.

Network Security

Network Security

We've discussed web applications, however, what supports the actual Internet? Network security is our final module of the course, where we're going to go through the procotols that explain how users communicate across a network. We'll also cover the attacks that exist against these protocols, and methods to protect against them.

25. Introduction to Networking

25. Introduction to Networking

To discuss network security, first we need to know how the network is designed. This section provides a (simplified) overview of the various Internet layers and how they interact. A video version of this section is available: see Lecture 11, Summer 2020.

25.1. Local Area Networks

The primary goal of the Internet is to move data from one location to another. A good analogy for the Internet is the postal system, which we'll refer to throughout this section.

The first building block we need is something that moves data across space, such as bits on a wire, radio waves, carrier pigeons, etc. Using our first building block, we can connect a group of local machines in a **local area network (LAN)**.

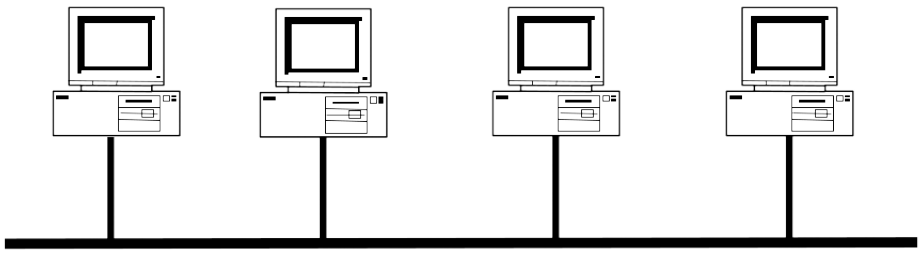

Figure 1: Diagram of a LAN, where computers are directly interconnected

Note that in a LAN, all machines are connected to all other machines. This allows any machine on the LAN to send and receive messages from any other machine on the same LAN. You can think of a LAN as an apartment complex, a local group of nearby apartments that are all connected. However, it would be infeasible to connect every machine in the world to every other machine in the world, so we introduce a **router** to connect multiple LANs.

A router is a machine that is connected to two or more LANs. If a machine wants to send a message to a machine on a different LAN, it sends the message to the router, which forwards the message to the second LAN. You can think of a router as a post office: to send a message somewhere outside of your local apartment complex, you'd take it to the post office, and they would forward your message to the other apartment complex.

With enough routers and LANs, we can connect the entire world in a **wide area network**, which forms the basis of the Internet.

25.2. Internet layering

You may have noticed that this design uses layers of abstraction to build the Internet. The lowest layer (layer 1, also called the physical layer) moves bits across space. Then, layer 2 (the link layer) uses layer 1

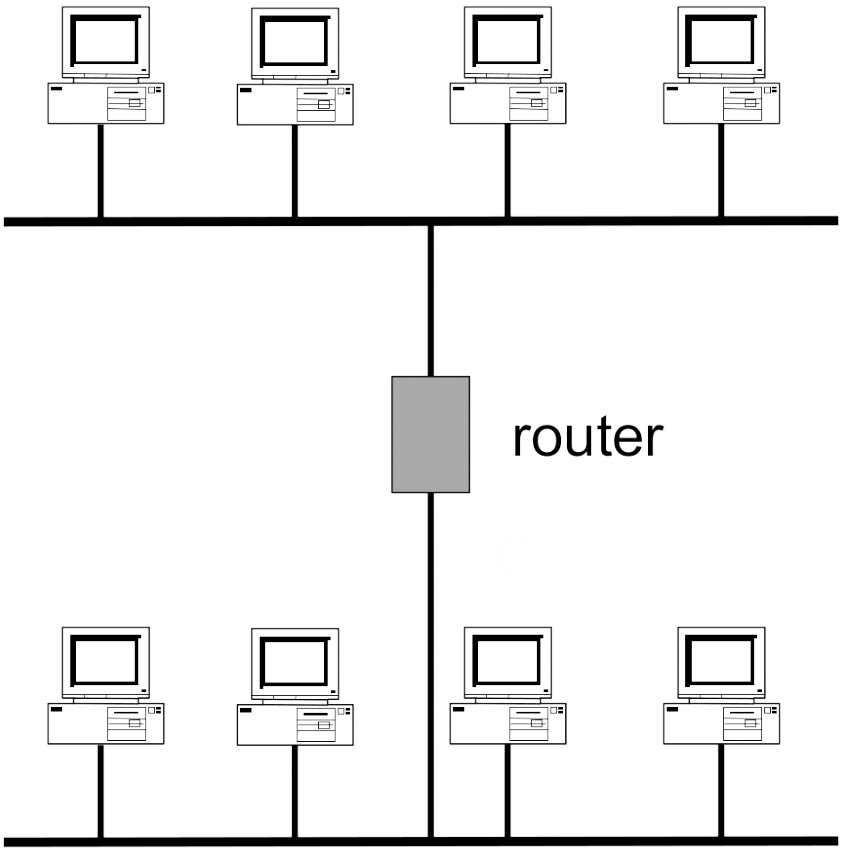

Figure 2: Diagram of a WAN, where two LANs are connected by a router

as a building block to connect local machines in a LAN. Finally, layer 3 (the internetwork layer) connects many layer 2 LANs. Each layer relies on services from a lower layer and provides services to a higher layer. Higher layers contain richer information, while lower layers provide the support necessary to send the richer information at the higher layers.

This design provides a clean abstraction barrier for implementation. For example, a network can choose to use wired or wireless communication at Layer 1, and the Layer 1 implementation does not affect any protocols at the other layers.

In total, there are 7 layers of the Internet, as defined by the OSI 7-layer model. However, this model is a little outdated, so some layers are obsolete, and additional layers for security have been added since then. We will see these higher layers later.

Layer	Name
7	Application
6.5	Secure Transport
6	*obsolete*
5	*obsolete*
4	Transport
3	(Inter)Network
2	Link
1	Physical

25.3. Protocols and Headers

Each layer has its own set of **protocols**, a set of agreements on how to communicate. Each protocol specifies how communication is structured (e.g. message format), how machines should behave while communicating (e.g. what actions are needed to send and receive messages), and how errors should be handled (e.g. a message timing out).

To support protocols, messages are sent with a **header**, which is placed at the beginning of the message and contains some metadata such as the sender and recipient's identities, the length of the message, identification numbers, etc. You can think of headers as the envelope of a letter: it contains the information needed to deliver the letter, and appears before the actual letter.

Because multiple protocols across different layers are needed to send a message, we need multiple headers on each packet. Each message begins as regular human-readable text (the highest layer). As the message is being prepared to get sent, it is passed down the protocol stack to lower layers (similar to how C programs are passed to lower layers to translate C code to RISC-V to machine-readable bits). Each layer adds its own header to the top of the message provided from the layer directly above. When the message reaches the lowest layer, it now has multiple headers, starting with the header for the lowest layer first.

Once the message reaches its destination, the recipient must unpack the message and decode it back into human-readable text. Starting at the lowest layer, the message moves up the protocol stack to higher layers. Each layer removes its header and provides the remaining content to the layer directly above. When the message reaches the highest layer, all headers have been processed, and the recipient sees the regular human-readable text from before.

25.4. Addressing: MAC, IP, Ports

Depending on the layer, a machine can be referred to by several different addresses.

Layer 2 (link layer) uses 48-bit (6-byte) **MAC addresses** to uniquely identify each machine on the LAN. This is not to be confused with MACs (message authentication codes) from the crypto section. Usually it is clear from context which type of MAC we are referring to, although sometimes cryptographic MACs are called MICs (message integrity codes) when discussing networking. MAC addresses are usually written as 6 pairs

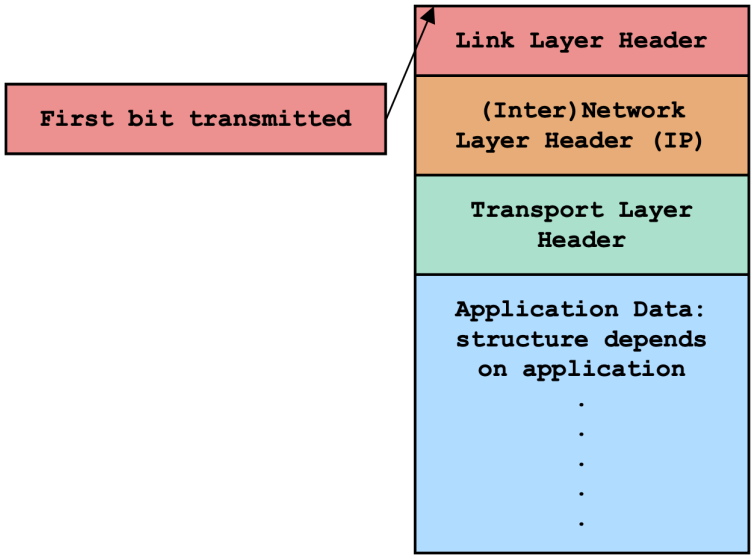

Figure 3: A diagram of a network packet structure, with the link layer header first, then the IP header, then the transport layer header, then the application data

of hex numbers, such as `ca:fe:f0:0d:be:ef`. There is also a special MAC address, the broadcast address of `ff:ff:ff:ff:ff:ff`, that says "send this message to everyone on the local network." You can think of MAC addresses as apartment numbers: they are used to uniquely identify people within one apartment complex, but are useless for uniquely identifying one person in the world. (Imagine sending a letter addressed to"Apartment 5." This might work if you're delivering letters within your own apartment complex, but how many Apartment 5s exist in the entire world?)

Layer 3 (IP layer) uses 32-bit (4-byte) **IP addresses** to uniquely identify each machine globally. IP addresses are usually written as 4 integers between 0 and 255, such as `128.32.131.10`. Because the Internet has grown so quickly, the most recent version of the layer 3 protocol, IPv6, uses 128-bit IP addresses, which are written as 8 2-byte hex values separated by colons, such as `cafe:f00d:d00d:1401:2414:1248:1281:8712`. However, for this class, you only need to know about IPv4, which uses 32-bit IP addresses.

Higher layers are designed to allow each machine to have multiple processes communicating across the network. For example, your computer only has one IP address, but it may have multiple browser tabs and applications open that all want to communicate over the network. To distinguish each process, higher layers assign each process on a machine a unique 16-bit **port number**. You can think of port numbers as room numbers: they are used to uniquely identify one person in a building.

The source and destination addresses are contained in the header of a message. For example, the Layer 2 header contains MAC addresses, the Layer 3 header contains IP addresses, and higher layer headers will contain port numbers.

25.5. Packets vs. Connections

Notice that in the postal system example, the post office has no idea if you and your pen pal are having a conversation through letters. The Internet is the same: at the physical, link, and internetwork layers, there is no concept of a connection. A router at the link layer only needs to consider each individual packet and send it to its destination (or, in the case of a long-distance message, forward it to another router somewhere closer to the destination). At the lower layers, we call individual messages **packets**. Packets are usually limited to a fixed length.

In order to actually create a two-way connection, we rely on higher layers, which maintain a connection by breaking up longer messages into individual packets and sending them through the lower layer protocols.

Higher-layer connections can also implement cryptographic protocols for additional security, as we'll see in the TLS section.

Note that so far, the Internet design has not guaranteed any correctness or security. Packets can be corrupted in transit or even fail to send entirely. The IP (Internet Protocol) at layer 3 only guarantees *best-effort delivery*, and does not handle any errors. Instead, we rely on higher layers for correctness and security.

25.6. Network Adversaries

Network adversaries can be sorted into 3 general categories. They are, from weakest to strongest:

Off-path Adversaries: cannot read or modify any packets sent over the connection.

On-path Adversaries: can read, but not modify packets.

In-path Adversaries: can read, modify, and block packets. Also known as a **man-in-the-middle**.

Note that all adversaries can send packets of their own, including faking or **spoofing** the packet headers to appear like the message is coming from somebody else. This is often as simple as setting the "source" field on the packet header to somebody else's address.

26. ARP

26. Wired Local Networks: ARP

26.1. Cheat sheet

- Layer: Link (2)
- Purpose: Translate IP addresses to MAC addresses
- Vulnerability: On-path attackers can see requests and send spoofed malicious responses
- Defense: Switches, arpwatch

26.2. Networking background: Ethernet

Recall that on a LAN (local-area network), all machines are connected to all other machines. Ethernet is one particular LAN implementation that uses wires to connect all machines.

Ethernet started as a broadcast-only network. Each node on the network could see messages sent by all other nodes, either by being on a common wire or a network **hub**, a simple repeater that took every packet it received and rebroadcast it to all the outputs. A receiver is simply supposed to ignore all packets not sent to either the receiver's MAC or the broadcast address. But this is only enforced in software, and most Ethernet devices can enter **promiscuous mode**, where it will receive all packets. This is also called **sniffing packets**.

For versions of Ethernet that are inherently broadcast, such as a hub, an adversary in the local network can see all network traffic and can also introduce any traffic they desire by simply sending packets with a spoofed MAC address. Sanity check: what type of adversary does this make someone on the same LAN network as a victim?[1]

26.3. Protocol: ARP

ARP, the **Address Resolution Protocol**, translates Layer 3 IP addresses into Layer 2 MAC addresses.

Say Alice wants to send a message to Bob, and Alice knows that Bob's IP address is `1.1.1.1`. The ARP protocol would follow three steps:

1. Alice would broadcast to everyone else on the LAN: "What is the MAC address of `1.1.1.1`?"
2. Bob responds by sending a message only to Alice: "My IP is `1.1.1.1` and my MAC address is `ca:fe:f0:0d:be:ef`." Everyone else does nothing.
3. Alice caches the IP address to MAC address mapping for Bob.

If Bob is outside of the LAN, then the router would respond in step 2 with its MAC address.

Any received ARP replies are always cached, even if no broadcast request (step 1) was ever made.

[1] A: On-path

26.4. Attack: ARP Spoofing

Because there is no way to verify that the reply in step 2 is actually from Bob, it is easy to attack this protocol. If Mallory is able to create a spoofed reply and send it to Alice before Bob can send his legitimate reply, then she can convince Alice that a different MAC address (such as Mallory's) corresponds to Bob's IP address. Now, when Alice wants to send a local message to Bob, she will use the malicious cached IP address to MAC address mapping, which might map Bob's IP address to Mallory's MAC address. This will cause messages intended for Bob to be sent to Mallory. Sanity check: what type of adversary is Mallory after she executes an ARP spoof attack?[2]

ARP spoofing is our first example of a race condition, where the attacker's response must arrive faster than the legitimate response to fool the victim. This is a common pattern for on-path attackers, who cannot block the legitimate response and thus must race to send their response first.

26.5. Defenses: Switches

A simple defense against ARP spoofing is to use a tool like arpwatch, which tracks the IP address to MAC address pairings across the LAN and makes sure nothing suspicious happens.

Modern wired Ethernet networks defend against ARP spoofing by using **switches** rather than hubs. Switches have a MAC cache, which keeps track of the IP address to MAC address pairings. If the packet's IP address has a known MAC in the cache, the switch just sends it to the MAC. Otherwise, it broadcasts the packet to everyone. Smarter switches can filter requests so that not every request is broadcast to everyone.

Higher-quality switches include **VLAN**s (Virtual Local Area Networks), which implement isolation by breaking the network into separate virtual networks.

[2] A: Man-in-the-middle. She can receive messages from Alice, modify them, then send them to Bob.

27. DHCP

27. DHCP

27.1. Cheat sheet

- Layer: 2-3 (see below)
- Purpose: Get configurations when first connecting to a network
- Vulnerability: On-path attackers can see requests and send spoofed malicious responses
- Defense: Accept as a fact of life and rely on higher layers

27.2. Protocol: DHCP

DHCP (**Dynamic Host Configuration Protocol**) is responsible for setting up configurations when a computer first joins a local network. These settings enable communication over LANs and the Internet, so it is sometimes considered a layer 2-3 protocol. The Internet layers are defined primarily for communication, so setup protocols like DHCP don't fit cleanly into the abstraction barriers in the layering model.

In order to connect to a network, you need a few things:

- An IP address, so other people can contact you
- The IP address of the DNS server, so you can translate a site name like www.google.com into an IP address (DNS is covered in more detail later)
- The IP address of the router (also called the **gateway**), so you can contact others on the Internet

The DHCP handshake follows four steps, between you (the client) and the server (who can give you the needed IP addresses)

1. **Client Discover**: The client broadcasts a request for a configuration.
2. **Server Offer**: Any server able to offer IP addresses responds with some configuration settings. (In practice, usually only one server replies here.)
3. **Client Request**: The client broadcasts which configuration it has chosen.
4. **Server Acknowledge**: The chosen server confirms that its configuration has been chosen.

The configuration information provided in step 2 (server offer) is sometimes called a **DHCP lease**. The offer may include a lease time. After the time expires, the client must ask to renew the lease to keep using that configuration, or else the DHCP server will free up those settings for other devices that request leases later.

Notice that both client messages are broadcast. Step 3 (client request) must be broadcast so that if multiple servers made offers in step 2, all the servers know which one has been chosen. Sanity check: why must client discover be broadcast?[1]

[1] A: Before DHCP, the client has no idea where the servers are.

27.3. Networking background: NAT

Because there are more computers than IPv4 addresses on the modern Internet, and not all networks support IPv6 (expanded address space) yet, DHCP supports **NAT (Network Address Translation)**, which allows multiple computers on a local network to share an IP address. When a computer requests a configuration through DHCP, the router (DHCP server) assigns that computer a placeholder IP address. This address usually comes from a reserved block of private IP addresses that are invalid on the Internet, but can be used as placeholders in the local network.

When a computer sends a packet to the Internet, the packet passes through the router first. The router stores a record mapping the internal (source) IP address to the remote (destination) IP address, for processing potential replies. Then the router replaces the placeholder IP address with a valid IP address, and sends the packet to the remote sever on the Internet. Sanity check: does this replacement happen for the source or destination IP address?[2] When the router sees an incoming packet, it checks the stored mappings, converts the destination IP address back to the correct placeholder address, and forwards the message to the original computer on the local network. With NAT, the router could potentially use a single valid IP address to send packets on behalf of every computer on the local network.

27.4. Attack

The attack on DHCP is almost identical to ARP spoofing. At the server offer step, an attacker can send a forged configuration, which the client will accept if it arrives before the legitimate configuration reply. The attacker could offer its own IP address as the gateway address, which makes the attacker a man-in-the-middle. Packets intended for the network would be sent to the attacker, who can modify them before forwarding them to the real gateway. The attacker can also become a man-in-the-middle by manipulating the DNS server address, which lets the attacker supply malicious translations between human-readable host names (www.google.com) and IP addresses (6.6.6.6).

27.5. Defenses

In reality, many networks just accept DHCP spoofing as a fact of life and rely on the higher layers to defend against attackers (the general idea: if the message sent is properly encrypted, the man-in-the-middle can't do anything anyway).

Defending against low-layer attacks like DHCP spoofing is hard, because there is no trusted party to rely on when we're first connecting to the network.

[2] Source, since it's an outgoing packet.

28. WPA

28. Wireless Local Networks: WPA2

28.1. Cheat sheet

- Layer: Link (2)
- Purpose: Communicate securely in a wireless local network
- Vulnerability: On-path attackers can learn the encryption keys from the handshake and decrypt messages (includes brute-forcing the password if they don't know it already)
- Defense: WPA2-Enterprise

28.2. Networking background: WiFi

Another implementation of the link layer is WiFi, which wirelessly connects machines in a LAN. Because it uses wireless connections over cellular networks, WiFi has some differences from wired Ethernet, but these are out of scope for this class. For the purposes of this class, WiFi behaves mostly like Ethernet, with the same packet format and similar protocols like ARP for address translation.

To join a WiFi network, your computer establishes a connection to the network's **AP (Access Point)**. Generally the AP is continuously broadcasting beacon packets saying "I am here" and announcing the name of the network, also called the **SSID (Service Set Identifier)**. When you choose to connect to a WiFi network (or if your computer is configured to automatically join a WiFi network), it will broadcast a request to join the network.

If the network is configured without a password, your computer immediately joins the network, and all data is transmitted without encryption. This means that anybody else on the same network can see your traffic and inject packets, like in ARP spoofing.

28.3. Protocol

WPA2-PSK (WiFi Protected Access: Pre-Shared Key) is a protocol that enables secure communications over a WiFi network by encrypting messages with cryptography.

In WPA2-PSK, a network has one password for all users (this is the WiFi password you ask your friends for). The access point derives a **PSK (Pre-Shared Key)** by applying a password-based key derivation function (PBKDF2-SHA1) on the SSID and the password. Recall from the cryptography unit that password-based key derivation functions are designed to be slower by a large constant factor to make brute-force attacks more difficult. Sanity check: Why might we choose to include the SSID as input to the key derivation function?[1]

When a computer (client) wants to connect to a network protected with WPA2-PSK, the user must first type in the WiFi password. Then, the client uses the same key derivation function to generate the PSK. Sanity check: Why can't we be done here and use the PSK to encrypt all further communications?[2]

[1] By including the SSID, two different networks with the same password will still have different PSKs.
[2] Because everyone on the network would use the same PSK, so others on the same network can still decrypt your traffic.

To give each user a unique encryption key, after both the client and the access point independently derive the PSK, they participate in a handshake to generate shared encryption keys.

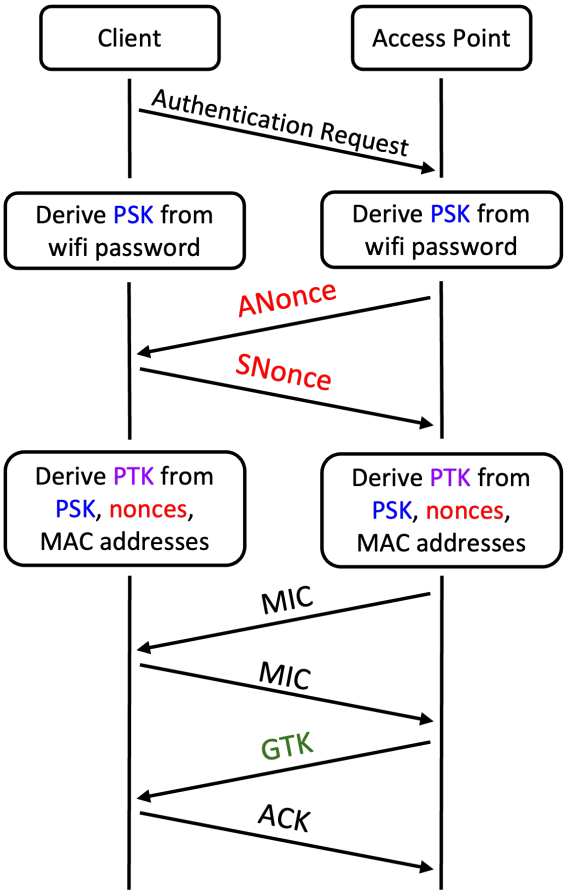

Figure 1: Diagram of the WPA2 handshake

1. The client and the access point exchange random nonces, the ANonce and the SNonce. The nonces ensure that different keys will be generated during each handshake. The nonces are sent without any encryption.
2. The client and access point independently derive the **PTK (Pairwise Transport Keys)** as a function of the two nonces, the PSK, and the MAC addresses of both the access point and the client.
3. The client and the access point exchange MICs (recall that these are MACs from the crypto unit) to check that no one tampered with the nonces, and that both sides correctly derived the PTK.
4. The access point encrypts the **GTK (Group Temporal Key)** and sends it to the client.
5. The client sends an ACK (acknowledgement message) to indicate that it successfully received the GTK.

Once the handshake is complete, all further communication between the client and the access point is encrypted with the PTK.

The GTK is used for messages broadcast to the entire network (i.e. sent to the broadcast MAC address, `ff:ff:ff:ff:ff:ff`). The GTK is the same for everyone on the network, so everyone can encrypt/send and decrypt/receive broadcast messages.

In practice, the handshake is optimized into a 4-way handshake, requiring only 4 messages to be exchanged between the client and the access point.

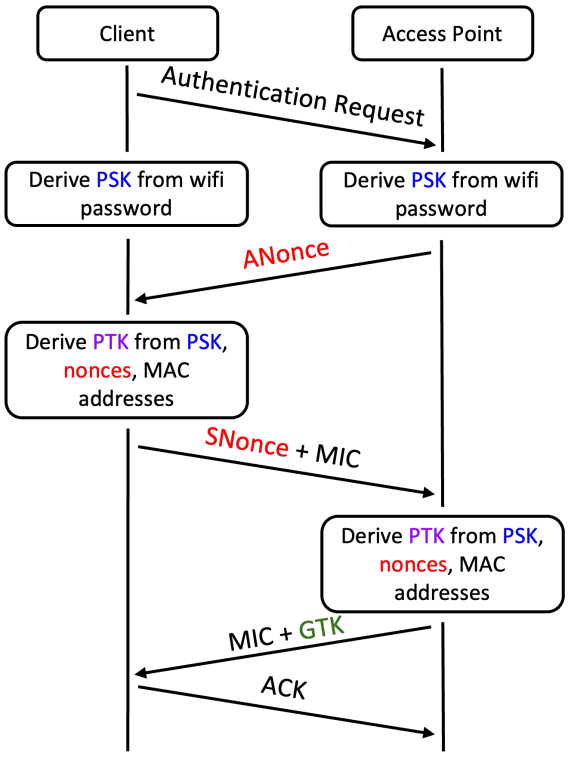

Figure 2: Diagram of the optimized WPA2 handshake used in practice

1. The access point sends the ANonce, as before.
2. Once the client receives the ANonce, it has all the information needed to derive the PTK, so it derives the PTK first. Then it sends the SNonce and the MIC to the access point.
3. Once the access point receives the SNonce, it can derive the PTK as well. Then it sends the encrypted GTK and the MIC to the client.
4. The client sends an ACK to indicate that it successfully received the GTK, as before.

28.4. Attacks

In the WPA2 handshake, everything except the GTK is sent unencrypted. Recall that the PTK is derived with the two nonces, the PSK, and the MAC addresses of both the access point and the client. This means that an on-path attacker who eavesdrops on the entire handshake can learn the nonces and the MAC addresses. If the attacker is part of the WiFi network (i.e. they know the WiFi password and generated the PSK), then they know everything necessary to derive the PTK. This attacker can decrypt all messages and eavesdrop on communications, and encrypt and inject messages.

Even if the attacker isn't on the WiFi network (doesn't know the WiFi password and cannot generate the PSK), they can try to brute-force the WiFi password. For each guessed password, the attacker derives the PSK from that password, uses the PSK (and the other unencrypted information from the handshake) to derive the PTK, and checks if that PTK is consistent with the MICs. If the WiFi password is low-entropy, an attacker with enough compute power can brute-force the password and learn the PTK.

28.5. Defenses: WPA2-Enterprise

The main problem leading to the attacks in the previous section is that every user on the network uses the same secrecy (the WiFi password) to derive private keys. To solve this, each user needs an different,

unique source of secrecy. This modified protocol is called **WPA2-Enterprise**. AirBears2 is an example of WPA2-Enterprise that you might be familiar with.

Instead of using one WiFi password for all users, WPA2-Enterprise gives authorized users a unique username and password. In WPA2-Enterprise, before the handshake occurs, the client connects to a secure authentication server and proves its identity to that server by providing a username and password. (The connection to the authentication server is secured with TLS, which is covered in a later section.) If the username and password are correct, the authentication server presents both the client and the access point with a random **PMK (Pairwise Master Key)** to use instead of the PSK. The handshake proceeds as in the previous section, but it uses the PMK (unique for each user) in place of the PSK (same for all users) to derive the PTK.

WPA-2 Enterprise defends against the attacks from the previous section, because the PMK is created randomly by a third-party authentication server and sent over encrypted channels to both the AP and the client. However, note that WPA2-Enterprise is still vulnerable against another authenticated user who executes an ARP or DHCP attack to become a man-in-the-middle.

29. BGP

29. IP Routing: BGP

29.1. Cheat sheet

- Layer: 3 (inter-network)
- Purpose: Send messages globally by connecting lots of local networks
- Vulnerability: Malicious local networks can read messages in intermediate transit and forward them to the wrong place
- Defense: Accept as a fact of life and rely on higher layers

29.2. Networking background: Subnets

Recall that IP addresses uniquely identify a single machine on the global network. (With NAT, the address could correspond to multiple machines, but this can be abstracted away when discussing IP.) When sending packets to a remote IP on a different local network, the packet must make many hops across many local networks before finally reaching its destination.

IP routes by "subnets", groups of addresses with a common prefix. A subnet is usually written as a prefix followed by the number of bits in the prefix. For example, `128.32/16` is an IPv4 subnet with all addresses beginning with the 16-bit prefix `128.32`. There are 2^{16} addresses on this prefix, because there are 32 - 16 = 16 bits not in the prefix. Sanity check: how many addresses are in the `128.32.131/24` subnet?[1] Routing generally proceeds on a subnet rather than individual IP basis.

There are some special reserved IP addresses and network blocks that do not represent machines and subnets. `127.0.0/24` and `::1` are "localhost", used to create 'network' connections to your own system. Also, `255.255.255.255` is the IPv4 broadcast address, sending to all computers within the local network.

When a client gets its configuration from DHCP, it is told its own IP address, the address of the gateway, and the size of the subnet it is on. To send a packet to another computer on the same local network, the client first verifies that the computer is on the same local network by checking that its IP address is in the same subnet (same IP prefix). Then, the client uses ARP to translate the IP address to a MAC address and directly sends the packet to that MAC address.

To send a packet to another computer on a different local network, the client sends the packet to the gateway, whose responsibility is to forward the packet towards the destination.

Past the gateway, the packet passes onto the general Internet, which is composed of many **ASs (Autonomous Systems)**, identified by unique **ASNs (Autonomous System Numbers)**. Each AS consists of one or more local networks managed by an organization, such as an Internet service provider (ISP), university, or business. Within each AS, packets can be routed by any mechanism the AS desires, usually involving a complicated set of preferences designed to minimize the AS's own cost.

[1] 2^8. The prefix is 24 bits, so there are 32 - 24 = 8 bits not in the prefix.

When an AS receives a packet, it first checks if that packet's final destination is located within the AS. If the final destination is within the AS, it routes the packet directly to the final destination. Otherwise, it must forward the packet to another AS that is closer to the final destination.

29.3. Protocol: BGP

Routing between ASs on the Internet is determined by BGP (the Border Gateway Protocol). BGP operates by having each AS advertise which networks it is responsible for to its neighboring ASs. Then each neighbor advertises that they can process packets to that network and provides information about the AS path that the packets would follow. The process continues until the entire Internet is connected into a graph with many paths between ASs. If an AS has a choice between two advertisements, it will generally select the shortest path. Actual BGP path selection is a fair bit more complicated than described here, but is out of scope for this class (take CS 168 to learn more).

29.4. Attack: Malicious ASs

The biggest problem with BGP is that it operates on trust, assuming that all ASs are effectively honest. Thus an AS can lie and say that it is responsible for a network it isn't, resulting in all traffic being redirected to the lying AS. There are further enhancements that allow a lying AS to act as a full man-in-the-middle, routing all traffic for a destination through the rogue AS.

Recall that IP operates on "best effort". Packets are delivered whole, but can be delivered in any order and may be corrupted or not sent at all. IPv4 and lower layers usually include checksums or CRC checks designed to detect corrupted packets. Sanity check: Why do the checksums not prevent a malicious AS from modifying packets?[2]

29.5. Defenses

In practice, there's not much anyone can do to defend against a malicious AS, since each AS operates relatively independently. Instead, we rely on protocols such as TCP at higher layers to guarantee that messages are sent. TCP will resend packets that are lost or corrupted because of malicious ASs. Also, cryptographic protocols at higher layers such as TLS can defend against malicious attackers, by guaranteeing confidentiality (attacker can't read the packets) and integrity (attacker can't modify the packets without detection) on packets. Both TCP and TLS are covered in later sections.

[2] Checksums are not cryptographic. The malicious AS could modify the packet and create a new checksum for the modified packet.

30. TCP and UDP

30. Transport Layer: TCP, UDP

30.1. Cheat sheet

- Layer: 4 (transport)
- Purpose: Establish connections between individual processes on machines (TCP and UDP). Guarantee that packets are delivered successfully and in the correct order (TCP only).
- Vulnerability: On-path and MITM attackers can inject data or RST packets. Off-path attackers must guess the 32-bit sequence number to inject packets.
- Defense: Rely on cryptography at a higher layer (TLS). Use randomly generated sequence numbers to stop off-path attackers.

30.2. Networking background: Ports

Recall that IP, the layer 3 (inter-network) protocol, is a best-effort protocol, meaning that packets can be corrupted, reordered, or dropped entirely. Also, IP addresses uniquely identify machines, but do not support multiple processes on one machine using the network (e.g. multiple browser tabs, multiple applications).

The transport layer solves the problem of multiple processes by introducing **port numbers**. Each process on a machine that wants to communicate over the network uses a unique 16-bit port number. Recall that port numbers are unique per machine, but cannot be used for global addressing–two machines can have processes with the same port number. However, an IP address and a port number together uniquely identify one process on one machine.

On client machines, such as your laptop, port numbers can be arbitrarily assigned. As long as each application uses a different port number, incoming packets can be sorted by port number and directed to the correct application. However, server machines offering services over the network need to use constant, well-known port numbers so client machines can send requests to those port numbers. For example, web servers always receive HTTP requests at port 80, and HTTPS (secure) requests at port 443. Ports below 1024 are "reserved" ports: only a program running as root can receive packets at those ports, but anyone can send packets to those ports.

The transport layer has 2 main protocols to choose from: TCP guarantees reliable, in-order packet delivery, while UDP does not. Both protocols use port numbers to support communication between processes. The choice of protocol depends on the context of the application.

30.3. Protocol: UDP

UDP (user datagram protocol) is a best-effort transport layer protocol. With UDP, applications send and receive discrete packets, and packets are not guaranteed to arrive, just like in IP. It is possible for datagrams to be larger than the underlying network's packet size, but this can sometimes introduce problems.

The UDP header contains 16-bit source and destination port numbers to support communication between processes. The header also contains a checksum (non-cryptographic) to detect corrupted packets.

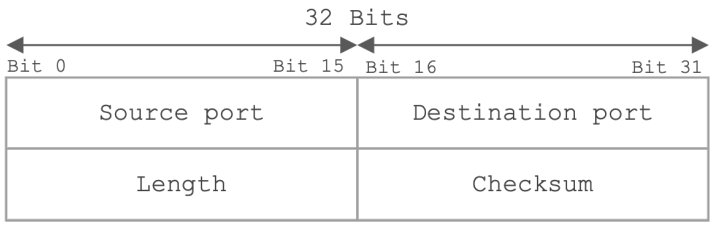

Figure 1: UDP header

30.4. Protocol: TCP

TCP (Transmission Control Protocol) is a reliable, in-order, connection-based stream protocol. In TCP, a client first establishes a connection to the server by performing a handshake. Once established, the connection is reliable and in order: TCP handles resending dropped packets until they are received on the other side and rearranging any packets received out of order. TCP also handles breaking up long messages into individual packets, which lets programmers think in terms of high-level, arbitrary-length bytestream connections and abstract away low-level, fixed-size packets.

Like UDP, the TCP header contains 16-bit source and destination port numbers to support communication between processes, and a checksum to detect corrupted packets. Additionally, a 32-bit **sequence number** and a 32-bit **acknowledgment (ACK) number** are used for keeping track of missing or out-of-order packets. Flags such as SYN, ACK, and FIN can be set in the header to indicate that the packet has some special meaning in the TCP protocol.

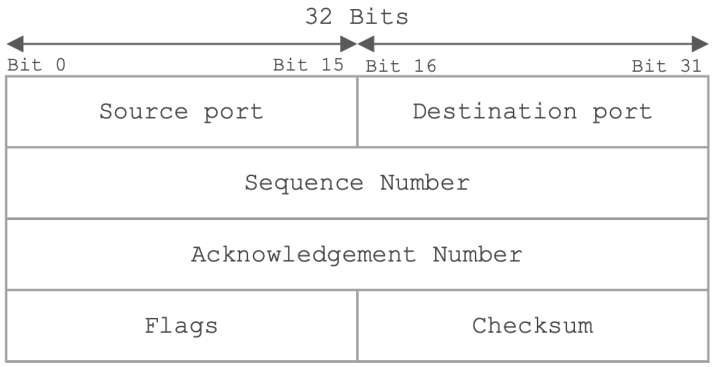

Figure 2: TCP header

A unique TCP connection is identified by a 5-tuple of (Client IP Address, Client Port, Server IP Address, Server Port, Protocol), where protocol is always TCP. In other words, a TCP connection is a sequence of back-and-forth communications between one port on one IP address, and another port on another IP address.

TCP communication works between any two machines, but it is most commonly used between a **client** requesting a service (such as your computer) and a **server** providing the service. To provide a service, the server waits for connection requests (sometimes called listening for requests), usually on a well-known port. To request the service, the client makes a connection request to that server's IP address and well-known port.

A TCP connection consists of two bytestreams of data: one from the client to the server, and one from the server to the client. The data in each stream is indexed using sequence numbers. Since there are two streams, there are two sets of sequence numbers in each TCP connection, one for each bytestream.

In every TCP packet, the sequence number field in the header is set to the index of the first byte sent in that packet. In packets from the client to the server, the sequence number is an index in the client-to-server bytestream, and in packets from the server to the client, the sequence number is an index in the server-to-client

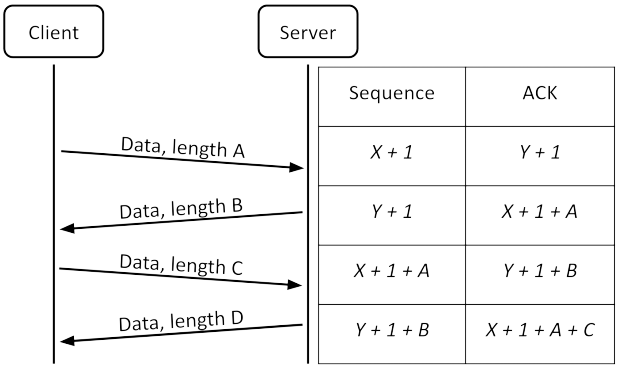

Figure 3: Diagram of TCP communication, with sequence numbers and ACK numbers

bytestream. If packets are reordered, the end hosts can use the sequence numbers to reconstruct the message in the correct order.

To ensure packets are successfully delivered, when one side receives a TCP packet, it must reply with an acknowledgment saying that it received the packet. If the packet was dropped in transit, the recipient will never send an acknowledgment, and after a timeout period, the sender will re-send that packet.

If the packet is delivered, but the acknowledgment is dropped in transit, the sender will notice that it never received an acknowledgment and will re-send the packet. The recipient will see a duplicate packet (since the original packet was delivered), discard the duplicate, and re-send the acknowledgment.

Sending acknowledgment packets is wasteful in a two-way communication, so TCP combines acknowledgment packets with data packets. Each TCP packet can contain both data and an acknowledgment that a previous packet was received.

To support acknowledgments, the acknowledgment (ACK) number in the header is set to the index of the last byte received, plus 1. (This is equivalent to the index of the next byte the sender expects to receive.) In other words, in packets from the client to the server, the ACK number is the next unsent byte in the server-to-client stream, and in packets from the server to the client, the ACK number is the next unsent byte in the client-to-server stream.

Note that in each packet, the sequence number is an index in the sender's bytestream, and the ACK number is an index in the recipient's bytestream.

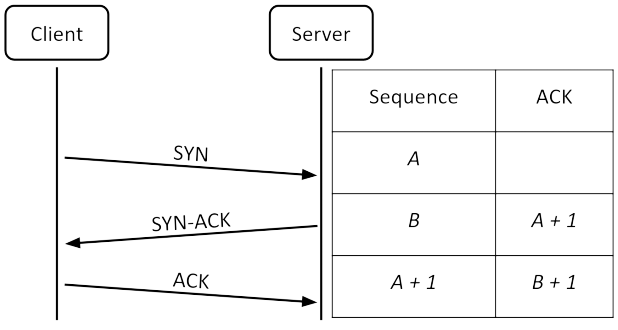

Figure 4: Diagram of the TCP 3-way handshake

Note that the sequence numbers do not start at 0 (for a security reason discussed below). Instead, to initiate a connection, the client and server participate in a three-way **TCP handshake** to exchange random initial sequence numbers.

1. The client sends a **SYN packet** (a packet with no data and the SYN flag set) to the server. The client sets the sequence number field to a random 32-bit **initial sequence number (ISN)**.
2. If the server decides to accept the request, it sends back a **SYN-ACK packet** (a packet with no data and both the SYN flag and ACK flag set). The server sets the sequence number field to its own random 32-bit initial sequence number (note that this is different from the client's ISN). The acknowledgment number is set to the client's initial sequence number + 1.
3. The client responds with an **ACK packet** (a packet with no data and the ACK flag set). The sequence number is set to the client's initial sequence number + 1 and the acknowledgement number is set to the server's initial sequence number + 1.

To end a connection, one side sends a FIN (a packet with the FIN flag set), and the other side replies with a FIN-ACK. This indicates that the side that sent the FIN will not send any more data, but can continue accepting data. This leaves the TCP connection in a "half closed" state, where one side stops sending but will receive and acknowledge further information. When the other side is done, it sends its own FIN as well, and it is acknowledged with a FIN-ACK reply.

Connections can also be unilaterally aborted. If one side sends a RST packet with a proper sequence number, this tells the other side that "I won't send any more data on this connection and I won't accept any more data on this connection." Unlike FIN packets, RST packets are not acknowledged. A RST usually indicates something went wrong, such as a program crashing or abruptly terminating a connection.

30.5. Tradeoffs between TCP and UDP

TCP is slower than UDP, because it requires a 3-way handshake at the start of each connection, and it will wait indefinitely for dropped packets to be sent again. However, TCP provides better correctness guarantees than UDP.

UDP is generally used when speed is a concern. For example, DNS requires extremely short response times, so it uses UDP instead of TCP at the transport layer. Video games and voice applications often use UDP because it is better to just miss a request than to stall everything waiting for a retransmission.

30.6. Attack: TCP Packet Injection

The main attack in TCP is **packet injection**. The attacker spoofs a malicious packet, filling in the header so that the packet looks like it came from someone in the TCP connection.

A related attack is **RST injection**. Instead of sending a packet with malicious data, the attacker sends a packet with the RST flag, causing the connection to abruptly terminate. This attack is useful for censorship: for example, Comcast used RST injection to abruptly terminate BitTorrent uploads.

Recall that that there are three types of network attackers. Each one has different capabilities in attacking the TCP protocol.

Off-path Adversary: The off-path adversary cannot read or modify any messages over the connection. Therefore, to attack a TCP communication, an off-path adversary must know or guess the values of the client IP, client port, server IP, and server port. Usually, the server IP address and port are well-known. Whether we know the client IP or port depends on our threat model. The off-path attacker must also guess the sequence number to inject a packet into the communication, because if the sequence number is too far off from what the recipient is expecting, it will reject the spoofed packet. Sanity check: What is the approximate probability of correctly guessing a random sequence number?[1]

On-path Adversary: The on-path adversary can read, but not modify messages. Since they can read the sequence numbers, IP addresses, and ports being used in the connection, an on-path adversary can inject messages into a TCP connection without guessing any values. As a concrete example, assume Alice has just sent a packet to Bob with sequence number X, and Bob responds with a packet of his own with sequence

[1]The sequence number is 32 bits, so guessing a random sequence number succeeds with probability $1/2^{32}$.

number Y and ACK X + 1. An on-path adversary Mallory wants to inject data into this TCP connection. While she cannot stop Alice from responding (because Mallory is not a man-in-the-middle), Mallory can race Alice's next packet with her own, using sequence number X + 1, ACK Y + 1, and Alice's IP and port. Since TCP on its own does not provide integrity, Bob will not be able to distinguish which message actually came from Alice, and which one came from Mallory.

In-path Adversary: The in-path (man-in-the-middle) adversary has all the powers of the on-path adversary and can additionally modify and block messages sent by either party. As a result, the same attack as the on-path adversary outlined above applies, and in addition, the in-path adversary doesn't have to race the party they are spoofing. A man in the middle can just block the message from ever arriving to the other party and send their own.

30.7. Defenses: TLS, random initial sequence numbers

The main problem here is that TCP by itself provides no confidentiality or integrity guarantees. To prevent injections like these, we rely on TLS, which is a higher-layer protocol that secures TCP communication with cryptography.

One important defense against off-path attackers is using random, unpredictable initial sequence numbers. This forces the off-path attacker to guess the correct sequence number with very low probability.

31. TLS

31. TLS

TLS (**Transport Layer Security**) is a protocol that provides an end-to-end encrypted communication channel. (You may sometimes see **SSL**, which is the old, deprecated version of TLS.) **End-to-end encryption** guarantees that even if any one part of the communication chain is compromised (for example, if the packet passes through a malicious AS), no one except the sender and receiver is able to read or modify the data being sent.

The original OSI 7-layer model did not consider security, so TLS is usually referred to as a layer 6.5 protocol. It is built on top of layer 4 TCP (layers 5 and 6 are obsolete), and it is used to provide secure communications to layer 7 applications. Examples of applications that use TLS are HTTP, which is renamed HTTPS if TLS is used; SMTP (Simple Mail Transport Protocol), which uses the STARTTLS command to enable TLS on emails; and **VPN** (Virtual Private Network) connections, which encrypt the user's traffic.

TLS relies on TCP to guarantee that messages are delivered reliably in the proper order. From the application viewpoint, TLS is effectively just like a TCP connection with additional security guarantees.

31.1. TLS Handshake

Because it's built on top of TCP, the TLS handshake starts with a TCP handshake. This lets us abstract away the notion of best-effort, fixed-size packets and think in terms of reliable messages for the rest of the TLS protocol.

The first message, `ClientHello`, presents a random number R_B and a list of encryption protocols it supports. The client can optionally also send the name of the server it actually wants to contact.

The second message, `ServerHello`, replies with its own random number R_S, the selected encryption protocol, and the server's **certificate**, which contains a copy of the server's public key signed by a **certificate authority** (**CA**).

If the client trusts the CA signing the certificate (e.g. that CA is included in the Chrome browser's pinned list of trusted CAs), then the client can use the signature to verify the server's public key is correct. If the client doesn't directly trust the CA, it may need to verify a chain of certificates in a PKI until it reaches the trusted root of the certificate chain. Either way, the client now has a trusted copy of the server's public key.

What is the public key being sent here? Every server implementing TLS must maintain a public/private key pair in order to support the PS exchange step you'll see next. We will assume that only the server knows the private key - if an attacker steals the private key, they would be able to impersonate the server, and the security guarantees no longer hold.

Sanity check: After the first two messages, can the client be certain that it is talking to the genuine server and not an impostor?[1]

The next step in TLS is to generate a random **Premaster Secret** (**PS**) known to only the client and the server. The PS should be generated so that no eavesdropper can determine the PS based on the data sent

[1] A: No. An attacker can obtain the genuine server's certificate by starting its own TLS connection with the genuine server, and then present a copy of that certificate in step 2.

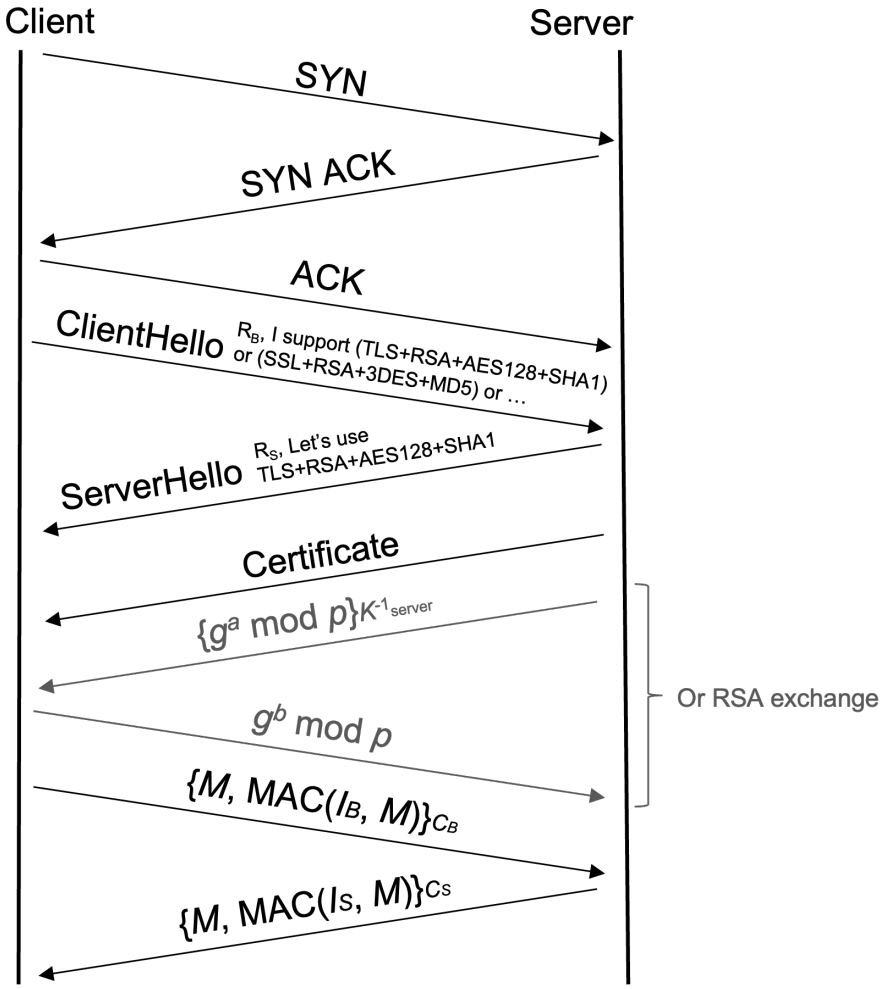

Figure 1: Diagram of the first part of the TLS handshake, from the ClientHello to the server certificate presentation

over the connection, and no one except the client and the legitimate server have enough information to derive the PS.

The first way to derive a shared PS is to encrypt it with RSA, shown in the last (blue) arrow here that depicts $\{PS\}_{K_{server}}$:

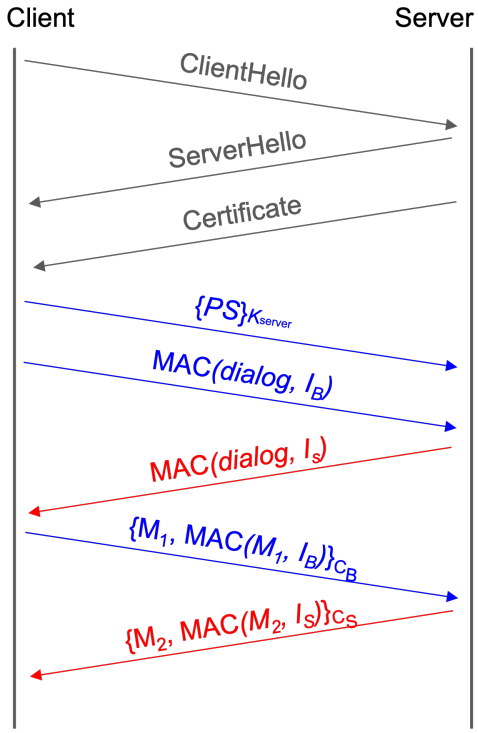

Figure 2: Diagram of the second part of the TLS handshake using RSA, from the server certificate presentation to the exchange of MACs

Here, the client generates the random PS, encrypts it with the server's public key, and sends it to the server, which decrypts using its private key.

Sanity check: How can the client be sure it's using the correct public key?[2]

We can verify that this method satisfies all the properties of a PS. Because it is encrypted when sent across the channel, no eavesdropper can decrypt and figure out its value. Also, only the legitimate server will be able to decrypt the PS (using its secret key), so only the client and the legitimate server will know the value of the PS.

The second way to generate a PS is to use Diffie-Hellman key exchange, shown in the fourth (red) and fifth (blue) arrows here that depict $\{g^a \bmod p\}_{K_{server}^{-1}}$ and $g^b \bmod p$ respectively:

The exchange looks just like classic Diffie-Hellman, except the server signs its half of the exchange with its secret key. The shared PS is the result of the key exchange, $g^{ab} \bmod p$.

Again, we can verify that this satisfies the properties of a PS. Diffie-Hellman's security properties guarantee that eavesdroppers cannot figure out PS, and no one but the client and the server know PS. We can be sure that the server is legitimate because the server's half of the key exchange is signed with its secret key.

An alternate implementation here is to use Elliptic Curve Diffie-Hellman (ECDHE). The specifics are out of scope, but it provides the same guarantees as regular DHE using elliptic curve math.

[2]A: It was signed by a certificate authority in the previous step.

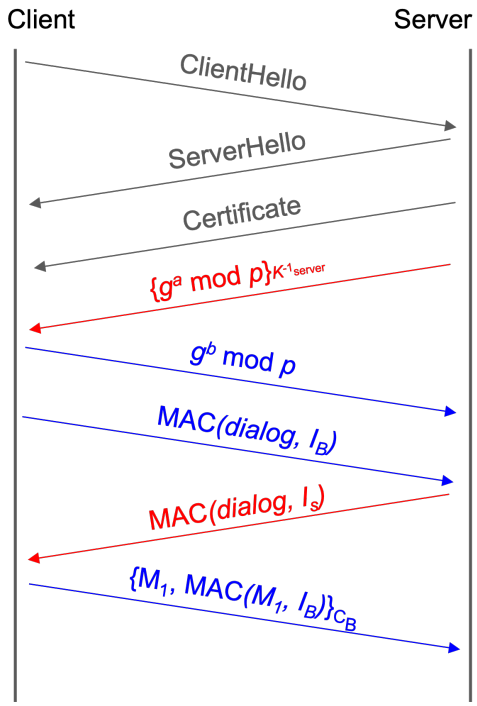

Figure 3: Diagram of the second part of the TLS handshake using Diffie-Hellman, from the server certificate presentation to the exchange of MACs

Generating the PS with DHE and ECDHE has a substantial advantage over RSA key exchange, because it provides **forward secrecy**. Suppose an attacker records lots of RSA-based TLS communications, and some time in the future manages to steal the server's private key. Now the attacker can decrypt PS values sent in old connections, which violates the security of those old TLS connections.

On the other hand, if the attacker steals the private key of a server using DHE or ECDHE-based TLS, they have no way of discovering the PS values of old connections, because the secrets required to generate the PS (a, b) cannot be discovered using the data sent over the connection $(g^a, g^b \bmod p)$. Starting from TLS 1.3, RSA key exchanges are no longer allowed for this reason.

Now that both client and server have a shared PS, they will each use the PS and the random values R_B and R_S to derive a set of four shared symmetric keys: an encryption key C_B and an integrity key I_B for the client, and an encryption key C_S and an integrity key I_S for the server.

Up until now, every message has been sent in plaintext over TLS. Sanity check: how might this be vulnerable?[3]

In order to ensure no one has tampered with the messages sent in the handshake so far, the client and server exchange and verify MACs over all messages sent so far. Notice that the client uses its own integrity key I_B to MAC the message, and the server uses its own integrity key I_S. However, both client and server know the value of I_B and I_S so that they can verify each other's MACs.

At the end of a proper TLS handshake, we have several security guarantees. (Sanity check: where in the handshake did these guarantees come from?)

1. The client is talking to the legitimate server.
2. No one has tampered with the handshake.
3. The client and server share a set of symmetric keys, unique to this connection, that no one else knows.

[3]A: TCP is insecure against on-path and MITM attackers, who can spoof messages.

Once the handshake is complete, messages are encrypted and MAC'd with the encryption and integrity keys of the sender before being sent. Because these messages have full confidentiality and integrity, TLS has achieved end-to-end security between the client and the server.

31.2. Replay attacks

Recall that a **replay attack** involves an attacker recording old messages and sending them to the server. Even though the attacker doesn't know what these messages decrypt to, if the protocol doesn't properly defend against replay attacks, the server might accept these messages as valid and allow the attacker to spoof a connection.

The public values R_B and R_S at the start of the handshake defend against replay attacks. To see why, let's assume that $R_B = R_S = 0$ every time and try to execute a replay attack on RSA-based TLS. Since the attacker is sending the same encrypted PS, and R_B and R_S are not changing, the server will re-generate the same symmetric keys. Now the attacker can replay messages from the old TLS connection, which will be accepted by the server because they have the correct MACs. Using new, randomly generated values R_B and R_S every time ensures that each connection results in a different set of symmetric keys, so replay attacks trying to establish a new connection with the same keys will fail.

What about a replay attack within the same connection? In practice, messages sent over TLS usually include some counter or timestamp so that an attacker cannot record a TLS message and send it again within the same connection.

31.3. TLS in practice

The biggest advantage and problem of TLS is the certificate authorities. "Trust does not scale", that is, you personally can't make trust decisions about everyone, but trust can be delegated, which is how TLS operates. We have delegated to a large number of companies, the **Certificate Authorities**, the responsibility of proving that a particular public key can speak for a particular site. This is what allows the system to work at all. But at the same time, unless additional measures are taken, this means that all CAs need to be trusted to speak for every site. This is why Chrome, for example, has a "pinned" CA list, so only some CAs are allowed to speak for certain websites.

Similarly, newer CAs implement **certificate transparency**, a mechanism where anyone can see all the certificates the CA has issued, implemented as a hash chain. Such CAs may issue a certificate incorrectly, but the impersonated victim can at least know this has happened. Certificates also expire and can be **revoked**, where a list of no-longer accepted certificates is published and regularly downloaded by a web browser or an online-service provides a mechanism to check if a particular certificate is revoked.

These days TLS is effectively free. The computational overhead is minor to the point of trivial: an ECDSA signature and ECDHE key exchange for the server, and such signatures and key exchanges are computationally minor: a single modern processor core can do tens of thousands of signatures or key exchanges per second. And once the key exchange is completed the bulk encryption is nearly free as most processors include routines specifically designed to accelerate AES.

This leaves the biggest cost of TLS in managing the private keys. Previously CAs charged a substantial amount to issue a certificate, but LetsEncrypt costs nothing because they have fully automated the process. You run a small program on your web server that generates keys, sends the public key to LetsEncrypt, and LetsEncrypt instructs that you put a particular file in a particular location on your server, acting to prove that you control the server. So LetsEncrypt has reduced the cost in two ways: It makes the TLS certificate monetarily free and, as important, makes it very easy to generate and use.

32. DNS

32. DNS

The Internet is commonly indexed in two different ways. Humans refer to websites using human-readable names such as `google.com` and `eecs.berkeley.edu`, while computers refer to websites using IP addresses such as `172.217.4.174` and `23.195.69.108`. **DNS**, or the **Domain Name System**, is the protocol that translates between the two.

32.1. Name servers

It would be great if there was single server that stored a mapping from every domain to every IP address that everyone could query, but unfortunately, there is no server big enough to store the IP address of every domain on the Internet and fast enough to handle the volume of DNS requests generated by the entire world. Instead, DNS uses a collection of many **name servers**, which are servers dedicated to replying to DNS requests.

Each name server is responsible for a specific zone of domains, so that no single server needs to store every domain on the Internet. For example, a name server responsible for the `.com` zone only needs to answer queries for domains that end in `.com`. This name server doesn't need to store any DNS information related to `wikipedia.org`. Likewise, a name server responsible for the `berkeley.edu` zone doesn't need to store any DNS information related to `stanford.edu`.

Even though it has a special purpose (responding to DNS requests), a name server is just like any other server you can contact on the Internet–each one has a human-readable domain name (e.g. `a.edu-servers.net`) and a computer-readable IP address (e.g. `192.5.6.30`). Be careful not to confuse the domain name with the zone. For example, this name server has `.net` in its domain, but it responds to DNS requests for `.edu` domains.

32.2. Name server hierarchy

You might notice two problems with this design. First, the `.com` zone may be smaller than the entire Internet, but it is still impractical for one name server to store all domains ending in `.com`. Second, if there are many name servers, how does your computer know which one to contact?

DNS solves both of these problems by introducing a new idea: when you query a name server, instead of always returning the IP address of the domain you queried, the name server can also direct you to another name server for the answer. This allows name servers with large zones such as `.edu` to redirect your query to other name servers with smaller zones such as `berkeley.edu`. Now, the name server for the `.edu` zone doesn't need to store any information about `eecs.berkeley.edu`, `math.berkeley.edu`, etc. Instead, the `.edu` name server stores information about the `berkeley.edu` name server and redirects requests for `eecs.berkeley.edu`, `math.berkeley.edu`, etc. to a `berkeley.edu` name server.

DNS arranges all the name servers in a tree hierarchy based on their zones:

The **root server** at the top level of the tree has all domains in its zone (this zone is usually written as `.`). Name servers at lower levels of the tree have smaller, more specific zones. Each name server is only responsible for storing information about their children, except for the name servers at the bottom of the tree, which are responsible for storing the actual mappings from domain names to IP addresses.

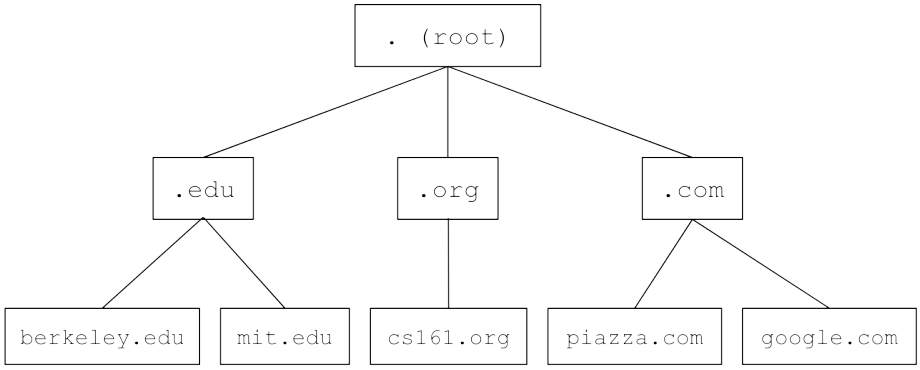

Figure 1: Diagram of an example DNS tree, with the root at the root, the top-level domains .edu, .org, and .com as the second level, and second-level domains such as berkeley.edu, cs161.org, and google.com at the third level

DNS queries always start at the root. The root will direct your query to one of its children name servers. Then you make a query to the child name server, and that name server redirects you to one of its children. The process repeats until you make a query to a name server at the bottom of the tree, which will return the IP address corresponding to your domain.

To redirect you to a child name server, the parent name server must provide the child's zone, human-readable domain name, and IP address, so that you can contact that child name server for more information.

As an example, a DNS query for `eecs.berkeley.edu` might have the following steps. (A comic version of this query is available at https://howdns.works/.)

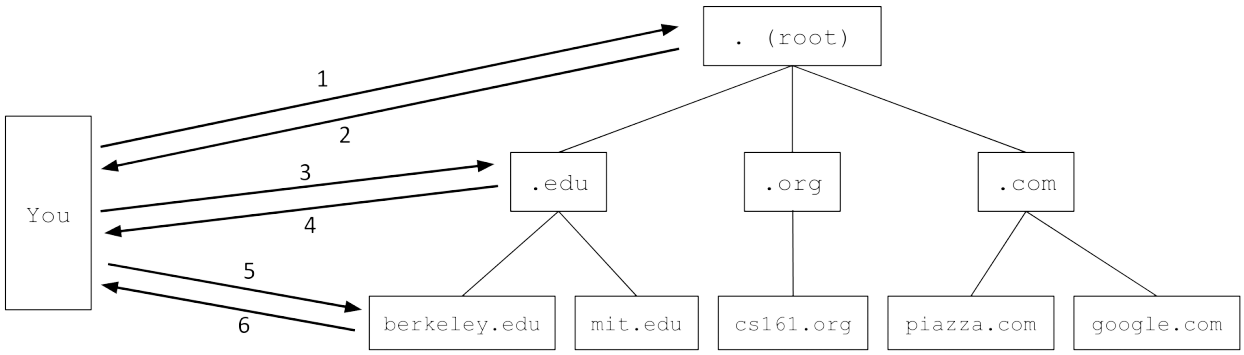

Figure 2: Diagram of a recursive DNS query, where your resolver queries the root nameserver first in query 1 and response 2, then the nameserver at the second level of the tree in query 3 and response 4, then a nameserver at the third level of the tree in query 5 and response 6

1. You to the root name server: Please tell me the IP address of `eecs.berkeley.edu`.
2. Root server to you: I don't know, but I can redirect you to another name server with more information. This name server is responsible for the `.edu` zone. It has human-readable domain name `a.edu-servers.net` and IP address `192.5.6.30`.
3. You to the `.edu` name server: Please tell me the IP address of `eecs.berkeley.edu`.
4. The `.edu` name server to you: I don't know, but I can redirect you to another name server with more information. This name server is responsible for the `berkeley.edu` zone. It has human-readable domain name `adns1.berkeley.edu` and IP address `128.32.136.3`.

5. You to the `berkeley.edu` name server: Please tell me the IP address of `eecs.berkeley.edu`.
6. The `berkeley.edu` name server to you: OK, the IP address of `eecs.berkeley.edu` is `23.185.0.1`.

A note on who is actually sending the DNS queries in this example: Your computer can manually perform DNS lookups, but in practice, your local computer usually delegates the task of DNS lookups to a **DNS Recursive Resolver** provided by your Internet service provider (ISP), which sends the queries, processes the responses, and maintains an internal cache of records. When performing a lookup, the **DNS Stub Resolver** on your computer sends a query to the recursive resolver, lets it do all the work, and receives the response. When thinking about DNS requests, you can usually focus on the messages being sent between the recursive resolver and the name server.

Congratulations, you now understand how DNS translates domains to IP addresses! The rest of this section describes the specific implementation details of DNS.

32.3 DNS Message Format

Since every website lookup must start with a DNS query, DNS is designed to be very lightweight and fast - it uses UDP (best-effort packets, no TCP handshakes) and has a fairly simple message format.

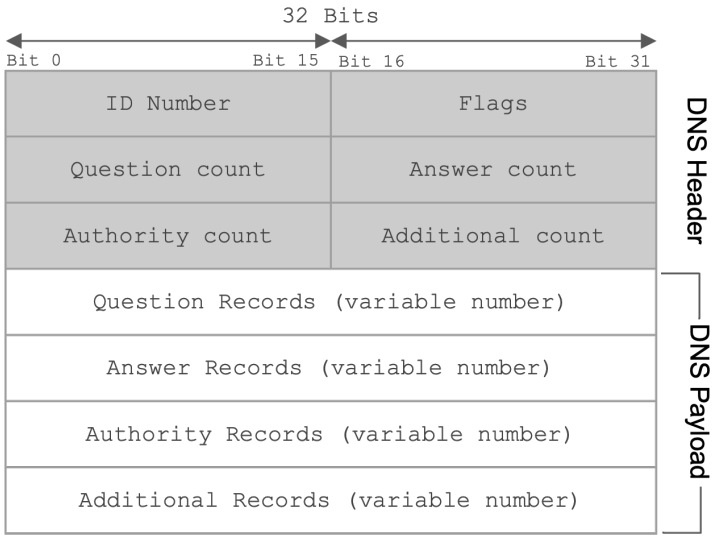

Figure 3: DNS packet

The first field is a 16 bit **identification field** that is randomly selected per query and used to match requests to responses. When a DNS query is sent, the ID field is filled with random bits. Since UDP is stateless, the DNS response must send back the same bits in the ID field so that the original query sender knows which DNS query the response corresponds to.

Sanity check: Which type(s) of adversary can read this ID field? Which type(s) of adversary cannot read the ID field and must guess it when attacking DNS?[1]

The next 16 bits are reserved for flags, which specify whether the message is a query or a response, as well as whether the query was successful (e.g. the `NOERROR` flag is set in the reply if the query succeeded, the `NXDOMAIN` flag is set in the reply if the query asked about a non-existent name).

The next field specifies the number of questions asked (in practice, this is always 1). The three fields after that are used in response messages and specify the number of **resource records** (RRs) contained in the message. We'll describe each of these categories of RRs in depth later.

[1] A: MITM and on-path can read the ID field. Off-path must guess the ID field.

The rest of the message contains the actual content of the DNS query/response. This content is always structured as a set of RRs, where each RR is a key-value pair with an associated type.

For completeness, a DNS record key is formally defined as a 3-tuple `<Name, Class, Type>`, where `Name` is the actual key data, `Class` is always `IN` for Internet (except for special queries used to get information about DNS itself), and `Type` specifies the record type. A DNS record value contains `<TTL, Value>`, where `TTL` is the time-to-live (how long, in seconds, the record can be cached), and `Value` is the actual value data.

There are two main types of records in DNS. **A type records** map domains to IP addresses. The key is a domain, and the value is an IP address. **NS type records** map zones to domains. The key is a zone, and the value is a domain.

Important takeaways from this section: Each DNS packet has a 16-bit random ID field, some metadata, and a set of resource records. Each record falls into one of four categories (question, answer, authority, additional), and each record contains a type, a key, and a value. There are A type records and NS type records.

32.4. DNS Lookup

Now, let's walk through a real DNS query for the IP address of `eecs.berkeley.edu`. You can try this at home with the `dig` utility–remember to set the `+norecurse` flag so you can unravel the recursion yourself.

Every DNS query begins with the root server. For redundancy, there are actually 13 root servers located around the world. We can look up the IP addresses of the root servers, which are public and well-known. In a real recursive resolver, these addresses are usually hardcoded.

The first root server has domain `a.root-servers.net` and IP address `198.41.0.4`. We can use `dig` to send a DNS request to this address, asking for the IP address of `eecs.berkeley.edu`.

```
$ dig +norecurse eecs.berkeley.edu @198.41.0.4

;; Got answer:
;; ->>HEADER<<- opcode: QUERY, status: NOERROR, id: 26114
;; flags: qr; QUERY: 1, ANSWER: 0, AUTHORITY: 13, ADDITIONAL: 27

;; QUESTION SECTION:
;eecs.berkeley.edu.          IN   A

;; AUTHORITY SECTION:
edu.                172800   IN   NS   a.edu-servers.net.
edu.                172800   IN   NS   b.edu-servers.net.
edu.                172800   IN   NS   c.edu-servers.net.
...

;; ADDITIONAL SECTION:
a.edu-servers.net.  172800   IN   A    192.5.6.30
b.edu-servers.net.  172800   IN   A    192.33.14.30
c.edu-servers.net.  172800   IN   A    192.26.92.30
...
```

In the first section of the answer, we can see the header information, including the ID field (`26114`), the return flags (`NOERROR`), and the number of records returned in each section.

The **question section** contains 1 record (you can verify by seeing `QUERY: 1` in the header). It has key `eecs.berkeley.edu`, type `A`, and a blank value. This represents the domain we queried for (the value is blank because we don't know the corresponding IP address).

The **answer section** is blank (`ANSWER: 0` in the header), because the root server didn't provide a direct answer to our query.

The **authority section** contains 13 records. The first one has key `.edu`, type `NS`, and value `a.edu-servers.net`. This is the root server giving us the zone and the domain name of the next name server we should contact. Each record in this section corresponds to a potential name server we could ask next.

The **additional section** contains 27 records. The first one has key `a.edu-servers.net`, type `A`, and value `192.5.6.30`. This is the root server giving us the IP address of the next name server by mapping a domain from the authority section to an IP address.

Together, the authority section and additional section combined give us the zone, domain name, and IP address of the next name server. This information is spread across two sections to maintain the key-value structure of the DNS message.

For completeness: `172800` is the TTL (time-to-live) for each record, set at 172,800 seconds = 48 hours here. The `IN` is the Internet class and can basically be ignored. Sometimes you will see records of type `AAAA`, which correspond to IPv6 addresses (the usual `A` type records correspond to IPv4 addresses).

Sanity check: What name server do we query next? How do we know where that name server is located? What do we query that name server for?[2]

```
$ dig +norecurse eecs.berkeley.edu @192.5.6.30

;; Got answer:
;; ->>HEADER<<- opcode: QUERY, status: NOERROR, id: 36257
;; flags: qr; QUERY: 1, ANSWER: 0, AUTHORITY: 3, ADDITIONAL: 5

;; QUESTION SECTION:
;eecs.berkeley.edu.           IN   A

;; AUTHORITY SECTION:
berkeley.edu.        172800   IN   NS   adns1.berkeley.edu.
berkeley.edu.        172800   IN   NS   adns2.berkeley.edu.
berkeley.edu.        172800   IN   NS   adns3.berkeley.edu.

;; ADDITIONAL SECTION:
adns1.berkeley.edu.  172800   IN   A    128.32.136.3
adns2.berkeley.edu.  172800   IN   A    128.32.136.14
adns3.berkeley.edu.  172800   IN   A    192.107.102.142
...
```

The next query also has an empty answer section, with `NS` records in the authority section and `A` records in the additional section which give us the domains and IP addresses of name servers responsible for the `berkeley.edu` zone.

```
$ dig +norecurse eecs.berkeley.edu @128.32.136.3

;; Got answer:
;; ->>HEADER<<- opcode: QUERY, status: NOERROR, id: 52788
;; flags: qr aa; QUERY: 1, ANSWER: 1, AUTHORITY: 0, ADDITIONAL: 1

;; QUESTION SECTION:
;eecs.berkeley.edu.          IN   A

;; ANSWER SECTION:
eecs.berkeley.edu.  86400   IN   A   23.185.0.1
```

[2] Query `a.edu-servers.net`, whose location we know because of the records in the additional section. Query for the IP address of `eecs.berkeley.edu` just like before.

Finally, the last query gives us the IP address corresponding to `eecs.berkeley.edu` in the form of a single `A` type record in the answer section.

In practice, because the recursive resolver caches as many answers as possible, most queries can skip the first few steps and used cached records instead of asking root servers and high-level name servers like `.edu` every time. Caching helps speed up DNS, because fewer packets need to be sent across the network to translate a domain name to an IP address. Caching also helps reduce request load on the highest-level name servers.

32.5. DNS Security: Bailiwick

DNS is insecure against a malicious name server. For example, if a `berkeley.edu` name server was taken over by an attacker, it could send answer records that point to malicious IP addresses.

However, a more dangerous exploit is using the additional section to poison the cache with even more malicious IP addresses. For example, this malicious DNS response would cause the resolver to associate `google.com` with an attacker-owned IP address `6.6.6.6`.

```
$ dig +norecurse eecs.berkeley.edu @192.5.6.30

...
;; ADDITIONAL SECTION:
adns1.berkeley.edu.  172800   IN   A    128.32.136.3
www.google.com       999999   IN   A    6.6.6.6
...
```

To prevent any malicious name server from doing too much damage, resolvers implement **bailiwick checking**. With bailiwick checking, a name server is only allowed to provide records in its zone. This means that the `berkeley.edu` name server can only provide records for domains under `berkeley.edu` (not `stanford.edu`), the `.edu` name server can only provide records for domains under `.edu` (not `google.com`), and the root name servers can provide records for anything.

32.6. DNS Security: On-path attackers and off-path attackers

Against an on-path attacker, DNS is completely insecure - everything is sent over plaintext, so an attacker can read the request, construct a malicious response message with malicious records and the correct ID field, and race to send the malicious reply before the legitimate response. If the time-to-live (TTL) of the malicious records is set to a very high number, then the victim will cache those malicous records for a very long time.

For both on-path and off-path attackers, if the legitimate response arrives before the fake response, it is cached. Caching limits the attacker to only a few tries per week, because future requests for that domain can reference the cache, so no DNS queries are sent. Since off-path attackers must guess the ID field with a $1/2^{16}$ probability of success, and they only get a few tries per week, DNS was believed to be secure against off-path attackers, until Dan Kaminsky discovered a flaw in the DNS protocol in 2008. This attack was so severe that Kaminsky was awarded with a Wikipedia article.

32.7. DNS Security: Kaminsky attack

The Kaminsky attack relies on querying for nonexistent domains. Remember that the legitimate response for a nonexistent domain is an `NXDOMAIN` status with no other records, which means that nothing is cached! This allows the attacker to repeatedly race until they win, without having to wait for cached records to expire.

An attacker can now include malicious additional records in the fake response for the nonexistent `fake161.berkeley.edu`:

```
$ dig fake161.berkeley.edu

;; Got answer:
;; ->>HEADER<<- opcode: QUERY, status: NOERROR, id: 29439
```

```
;; flags: qr aa; QUERY: 1, ANSWER: 0, AUTHORITY: 1, ADDITIONAL: 1

;; QUESTION SECTION:
;fake161.berkeley.edu.        IN  A

;; ADDITIONAL SECTION:
berkeley.edu.     999999    IN  A   6.6.6.6
```

If the fake response arrives first, the resolver will cache the malicious additional record. Notice that this doesn't violate bailiwick checking, since the name server responsible for answering `fake161.berkeley.edu` can provide a record for `berkeley.edu`.

Now that the attacker can try as many times as they want, all that's left is to force a victim to make thousands of DNS queries for nonexistent domains. This can be achieved by tricking the victim into visiting a website that tries to load lots of nonexistent domains:

```
<img src="http://fake001.berkeley.edu/image.jpg"/>
<img src="http://fake002.berkeley.edu/image.jpg"/>
<img src="http://fake003.berkeley.edu/image.jpg"/>
...
```

This HTML snippet will cause the victim's browser to try and fetch images from `http://fake001.berkeley.edu/image.jpg`, `http://fake002.berkeley.edu/image.jpg`, etc. To fetch these images, the browser will first make a DNS request for the domains `fake001.berkeley.edu`, `fake002.berkeley.edu`, etc. For each request, if the legitimate response arrives before the malicious response, or if the off-path attacker incorrectly guesses the ID field, nothing is cached, so the attacker can immediately try again when the victim makes the next DNS request to the next non-existent domain.

The Kaminsky attack allows on-path attackers to race until their fake response arrives first and off-path attackers to race until they successfully guess the ID field. There is no way to completely eliminate the Kaminsky attack in regular DNS, although modern DNS protocols add **UDP source port randomization** to make it much harder.

Recall that UDP is a transport-layer protocol like TCP, so a UDP packet requires a source port and destination port. The destination port must be well-known and constant (in practice, it is always 53), so everyone can send UDP packets to the correct port on the name server. However, DNS doesn't specify what source port the resolver uses to send queries, so source port randomization uses a random 16-bit source port for each query. The name server must send the response packet back to the correct source port of the resolver, so it must include the source port number in the destination port field of the response. Now, an attacker must guess the 16-bit ID field and the 16-bit source port in order to successfully forge a response packet. This decreases an off-path attacker's probability of success to $1/2^{32}$, which is much harder, but certainly not impossible.

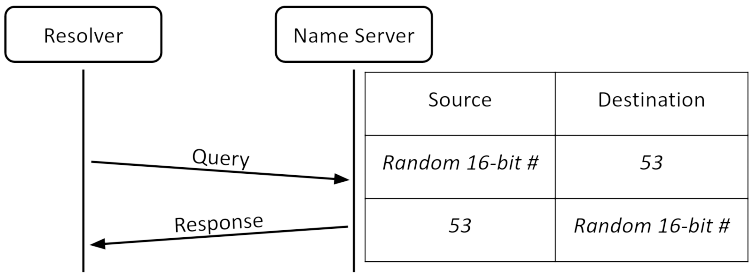

Figure 4: Diagram of source port randomization in use. The query's source port is randomized, and the destination port is 53. The response's source port is 53, and the destination port is the same randomized value

Sanity check: How much extra security does source port randomization provide against on-path attackers?[3]

[3]A: None, on-path attackers can see the source port value.

33. DNSSEC

33. DNSSEC

DNSSEC is an extension to regular DNS that provides integrity and authentication on all DNS messages sent. Sanity check: Why do we not care about the confidentiality of DNSSEC?[1]

33.1. Signing records

We want every DNS record to have integrity and authenticity, and we want everyone to be able to verify the integrity and authenticity of records. Digital signatures are a good fit in this situation, because only someone with the private key can create signatures, and everyone can use the public key to verify signatures.

To ensure integrity and authenticity, let's have every name server generate a public/private key pair and sign every record it sends with its private key. When the name server receives a DNS request, it sends the records, along with a signature on the records and the public key, to the resolver. The resolver uses the public key to verify the signature on the records.

Because of the signatures, a network attacker (MITM, on-path, off-path) cannot tamper with the data or inject malicious data without being detected (integrity). Also, the resolver can cache the signatures and the public key, and check at any time that the records actually came from the name server (authenticity).

You might see a flaw in this design: what if a name server is malicious? Then the malicious name server could return valid signatures on malicious records. How do we modify our design to prevent this?

33.2. Delegating trust

The main issue in our design so far is we lack a *trust anchor*. We want DNSSEC to defend against malicious name servers, so we cannot implicitly trust the name servers. However, if we don't trust anybody, then DNSSEC will never work (we'll never trust any records we get), so we must first choose a trust anchor, an entity that we implicitly trust. In DNSSEC, the root servers are the trust anchor: every computer automatically assumes that the root server is honest and uncompromised. In real life, this is a safe assumption, because the organizations overseeing the Internet hold painstakingly formal ceremonies to ensure that the root server is uncompromised. (If you're interested, you can read more about the root signing ceremony here.)

Given a trust anchor, we can now *delegate trust* from the trust anchor to somebody else. If the root endorses Alice, then you can be sure that Alice is trusted as well, since you implicitly trust the root. Also, if Alice endorses Bob, then you can be sure that Bob is trusted, since you trust Alice. This trust delegation starting from the root is how DNSSEC delegates trust from the root to all legitimate name servers, while protecting against malicious name servers.

Consider two parties, root and Alice, who each have a public key and a private key. You trust root, because it is the trust anchor. The root can delegate trust to Alice by *signing Alice's public key*. The root's signature on Alice's public key effectively says that Alice's public key is trustworthy, and the root trusts any message signed by Alice using her corresponding private key.

[1] A: DNS responses don't contain sensitive data. Anyone could query the name servers for the same information.

Now, when Alice signs a message, we can use Alice's public key to verify that the message was properly signed by Alice. Also, we know that Alice's public key is trusted, because the root has signed it, and we implicitly trust the root.

If Alice was malicious, then the root would not delegate trust to her by signing her public key, because we are trusting that the root is honest and uncompromised.

We can apply this delegation idea to the entire DNS tree. Each name server will sign the public key of all its trusted children name servers. For example, root signs `.edu`'s public key. We trust root, and root signed `.edu`'s public key, so now we trust `.edu`. Next, `.edu` signs `berkeley.edu`'s public key. We trust `.edu`, and `.edu` signed `berkeley.edu`'s public key, so now we trust `berkeley.edu`.

33.3. DNSSEC Intuition

With these ideas in mind, let's revisit the DNS query for `eecs.berkeley.edu` from earlier and convert it to a secure DNSSEC query. *The DNSSEC additions are italicized.*

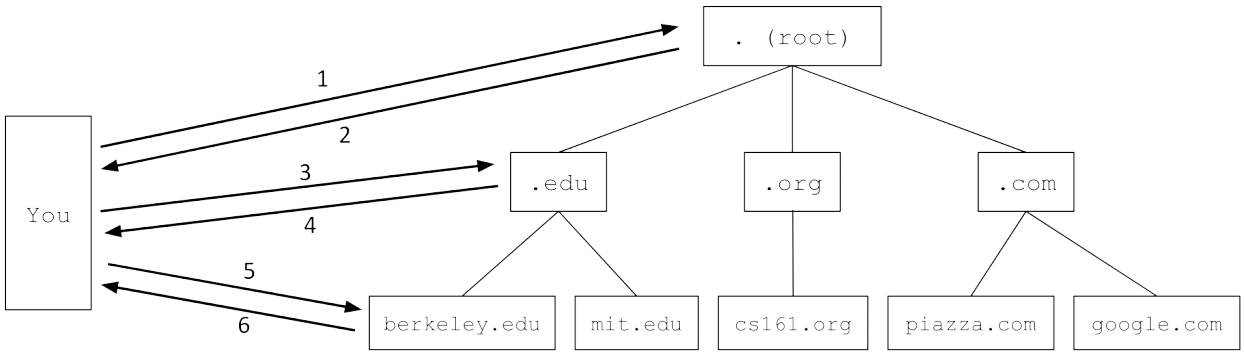

Figure 1: Diagram of a recursive DNS query, where your resolver queries the root nameserver first in query 1 and response 2, then the nameserver at the second level of the tree in query 3 and response 4, then a nameserver at the third level of the tree in query 5 and response 6

1. You to the root name server: Please tell me the IP address of `eecs.berkeley.edu`.
2. Root server to you: I don't know, but I can redirect you to another name server with more information. This name server is responsible for the `.edu` zone. It has human-readable domain name `a.edu-servers.net` and IP address `192.5.6.30`. *Here is a signature on the next name server's public key. If you trust me, then now you trust them too. Finally, here is my public key.*
3. You to the `.edu` name server: Please tell me the IP address of `eecs.berkeley.edu`.
4. The `.edu` name server to you: I don't know, but I can redirect you to another name server with more information. This name server is responsible for the `berkeley.edu` zone. It has human-readable domain name `adns1.berkeley.edu` and IP address `128.32.136.3`. *Here is a signature on the next name server's public key. If you trust me, then now you trust them too. Finally, here is my public key.*
5. You to the `berkeley.edu` name server: Please tell me the IP address of `eecs.berkeley.edu`.
6. The `berkeley.edu` name server to you: OK, the IP address of `eecs.berkeley.edu` is `23.185.0.1`. *Finally, here is my public key and a signature on the answer.*

Note that we implicitly trust all signed messages from the root, because the root is our trust anchor. In practice, all DNS resolvers have the root's public key hardcoded, and any messages verified with that hardcoded key are implicitly trusted.

Congratulations, you now have all the intuition for how DNSSEC works! The rest of this section shows how we implement this design in DNS.

33.4. New DNSSEC record types

To store cryptographic information in DNS messages, we need to introduce a few new record types.

The **`DNSKEY` type record** encodes a public key.

The **`RRSIG` type record** is a signature on a set of multiple other records in the message, all of the same type. For example, if the authority section returns 13 `NS` type records, you can sign all 13 records at once with one `RRSIG` type record. However, to sign the 26 `A` type records in the additional section, you would need another `RRSIG` type record. In addition to the actual cryptographic signature, the `RRSIG` type record contains the type of the records being signed, the signature creation and expiration date, and the identity of the signer (information about which public key/`DNSKEY` record should be used to verify this signature).

The **`DS` (Delegated Signer) type record** is a hash of the signer's name and a child's public key. The `DS` record, combined with a `RRSIG` record that signs the `DS` record, effectively allows each name server to sign the public key of its trusted children.

All DNSSEC cryptographic records additionally include some (uninteresting) metadata, such as which algorithm was used for signing/verifying/hashing.

You might have noticed that the number of additional records is always 1 more than the actual number of additional records that appear in the response. For example, consider the final query in our regular DNS query walkthrough:

```
$ dig +norecurse eecs.berkeley.edu @128.32.136.3

;; Got answer:
;; ->>HEADER<<- opcode: QUERY, status: NOERROR, id: 52788
;; flags: qr aa; QUERY: 1, ANSWER: 1, AUTHORITY: 0, ADDITIONAL: 1

;; QUESTION SECTION:
;eecs.berkeley.edu.          IN   A

;; ANSWER SECTION:
eecs.berkeley.edu.  86400   IN   A   23.185.0.1
```

The response reports 1 additional record but shows no additional records at all. This extra record corresponds to the `OPT` pseudosection (seen just above the question section). This pseudosection allows extra space for DNSSEC-specific flags (e.g. the `DO` flag requests DNSSEC information), but in order to be backwards-compatible with regular DNS, the section is encoded as an additional record when sent in the request and the reply.

33.5. Key Signing Keys and Zone Signing Keys

There is one final complication in DNSSEC–what if a name server wants to change its key pair? A key change is necessary if, for example, an attacker steals the private key of a trusted name server, because now the attacker can impersonate a trusted name server.

In our current DNSSEC design, a name server that wants to change keys must notify its parent name server so that the parent can change the DS record (which endorses the child's public key). As it turns out, this process is difficult to perform securely and can easily go wrong.

To minimize the use of this difficult key change protocol, each DNSSEC name server generates two public/private key pairs. The **key signing key (KSK)** is only used to sign the zone signing key, and the **zone signing key (ZSK)** is used to sign everything else.

In our previous design with one key pair, the name server sends (1) a set of records, (2) a signature on those records, and (3) the public key (endorsed by the parent). The DNS resolver uses the public key to verify the signature, and accepts the set of records.

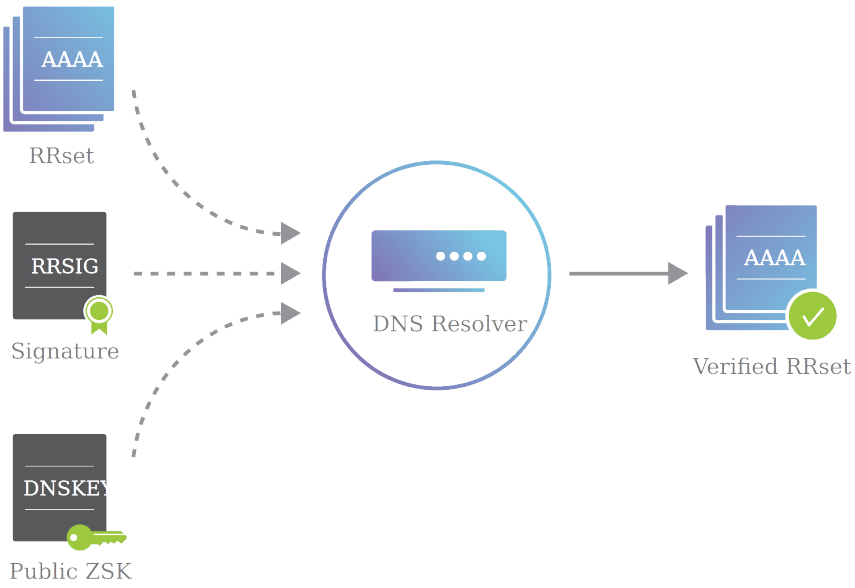

Figure 2: Diagram depicting a ZSK used to sign records

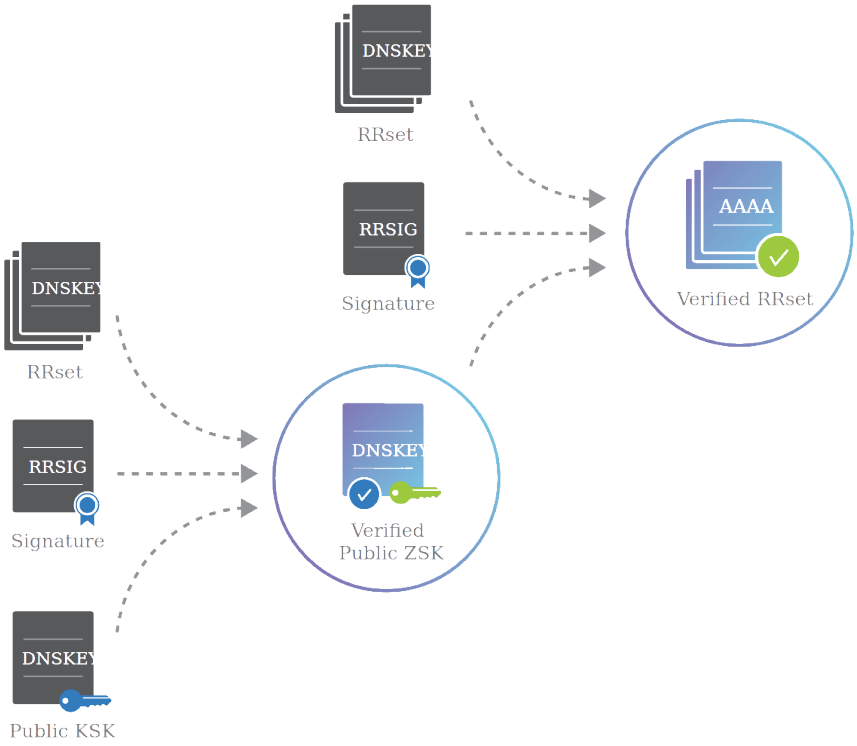

Figure 3: Diagram depicting a KSK used to sign a ZSK, which is then used to sign records

In our new design with two key pairs, the name server sends (1) the public ZSK, (2) a signature on the public ZSK, and (3) the public KSK (endorsed by the parent). The DNS resolver uses the public KSK to verify the signature, and accepts the public ZSK. Note that this is the exact same structure that was used to sign records before, but in this case, the record is the public ZSK, signed using the KSK.

Another way to think about this step is to recall that a parent endorses a child by signing its public key. You can think of the KSK as the "parent" and the ZSK as the "child," both within one name server. The parent (KSK) endorses the child (ZSK) by signing the public ZSK.

The result of this first step is that we now have a trusted public ZSK. The second step is the same as before: the name server sends a set of records, a signature on those records (using the private ZSK), and the public ZSK (endorsed by the KSK in the previous step).

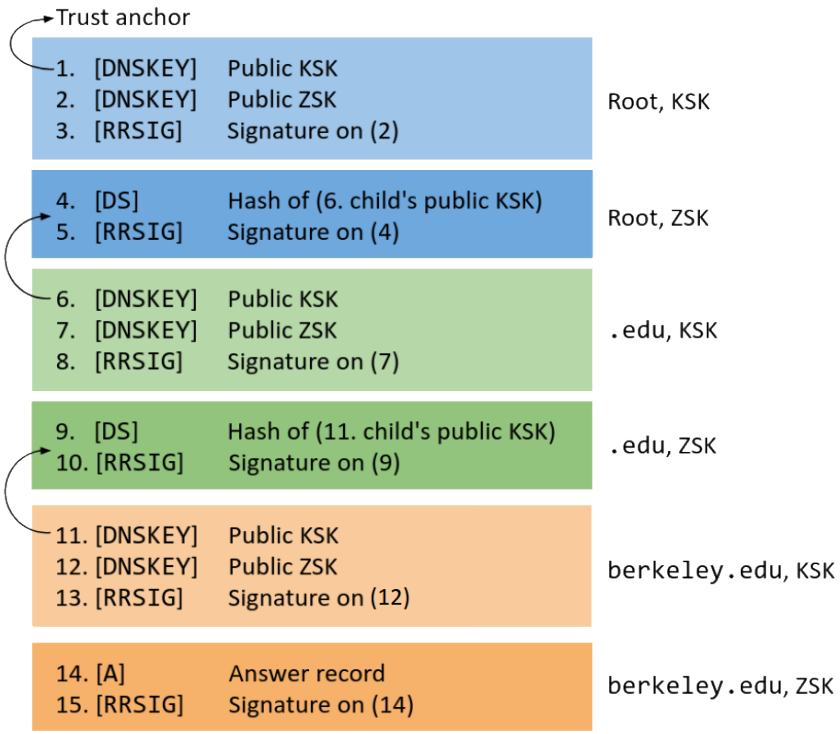

Figure 4: Diagram of the full chain of trust in DNSSEC. The trust anchor is the root's KSK, which is used to sign the root's ZSK, which is used to sign .edu's KSK, which is used to sign .edu's ZSK, which is used to sign berkeley.edu's KSK, which is used to sign berkeley.edu's ZSK, which is used to sign berkeley.edu's A record

Here is a diagram of the entire two-key DNSSEC. Each color (blue, green, orange) represents a name server. The lighter shade represents records signed with the KSK. The darker shade represents records signed with the ZSK.

Verification would proceed as follows.

- Light blue: Because of our trust anchor, we trust the KSK of the root (1). The root's KSK signs its ZSK, so now we trust the root's ZSK (2-3).
- Dark blue: We trust the root's ZSK. The root's ZSK signs `.edu`'s KSK (4-5), so now we trust `.edu`'s KSK.
- Light green: We trust the `.edu`'s KSK (6). `.edu`'s KSK signs `.edu`'s ZSK, so now we trust `.edu`'s ZSK (7-8).

- Dark green: We trust `.edu`'s ZSK. `.edu`'s ZSK signs `berkeley.edu`'s KSK (9-10), so now we trust `berkeley.edu`'s KSK.
- Light orange: We trust the `berkeley.edu`'s KSK (11). `berkeley.edu`'s KSK signs `berkeley.edu`'s ZSK, so now we trust `berkeley.edu`'s ZSK (12-13).
- Dark orange: We trust `berkeley.edu`'s ZSK. `berkeley.edu`'s ZSK signs the final answer record (14-15), so now we trust the final answer.

33.6. DNSSEC query walkthrough

Now we're ready to see a full DNSSEC query in action. As before, you can try this at home with the dig utility–remember to set the `+norecurse` flag so you can unravel the recursion yourself, and remember to set the `+dnssec` flag to enable DNSSEC.

First, we query the root server for its public keys. Recall that the root's IP address, `198.41.0.4`, is publicly-known and hardcoded.

```
$ dig +norecurse +dnssec DNSKEY . @198.41.0.4

;; Got answer:
;; ->>HEADER<<- opcode: QUERY, status: NOERROR, id: 7149
;; flags: qr aa; QUERY: 1, ANSWER: 3, AUTHORITY: 0, ADDITIONAL: 1

;; OPT PSEUDOSECTION:
; EDNS: version: 0, flags: do; udp: 1472
;; QUESTION SECTION:
;.              IN    DNSKEY

;; ANSWER SECTION:
.    172800    IN    DNSKEY    256 {ZSK of root}
.    172800    IN    DNSKEY    257 {KSK of root}
.    172800    IN    RRSIG     DNSKEY {signature on DNSKEY records}
...
```

In this response, the root has returned its public ZSK, public KSK, and a `RRSIG` type record over the two `DNSKEY` type records. We can use the public KSK to verify the signature on the public ZSK.

Because we implicitly trust the root's KSK (trust anchor), and the root's KSK signs its ZSK, we now trust the root's ZSK.

Next, we query the root server for the IP address of `eecs.berkeley.edu`.

```
$ dig +norecurse +dnssec eecs.berkeley.edu @198.41.0.4

;; Got answer:
;; ->>HEADER<<- opcode: QUERY, status: NOERROR, id: 5232
;; flags: qr; QUERY: 1, ANSWER: 0, AUTHORITY: 15, ADDITIONAL: 27

;; OPT PSEUDOSECTION:
; EDNS: version: 0, flags: do; udp: 4096
;; QUESTION SECTION:
;eecs.berkeley.edu.           IN   A

;; AUTHORITY SECTION:
edu.                 172800   IN   NS      a.edu-servers.net.
edu.                 172800   IN   NS      b.edu-servers.net.
edu.                 172800   IN   NS      c.edu-servers.net.
```

```
...
edu.                  86400    IN   DS      {hash of .edu's KSK}
edu.                  86400    IN   RRSIG   DS {signature on DS record}

;; ADDITIONAL SECTION:
a.edu-servers.net.    172800   IN   A       192.5.6.30
b.edu-servers.net.    172800   IN   A       192.33.14.30
c.edu-servers.net.    172800   IN   A       192.26.92.30
...
```

DNSSEC doesn't remove any records compared to regular DNS–the question, answer (blank here), authority, and additional sections all contain the same records from regular DNS. However, DNSSEC adds a `DS` record and a `RRSIG` signature record on the `DS` record. Together, these two records sign the KSK of the `.edu` name server with the root's ZSK. Since we trust the root's ZSK (from the previous step), now we trust the `.edu` name server's KSK.

Next, we query the `.edu` name server for its public keys.

```
$ dig +norecurse +dnssec DNSKEY edu. @192.5.6.30

;; Got answer:
;; ->>HEADER<<- opcode: QUERY, status: NOERROR, id: 9776
;; flags: qr aa; QUERY: 1, ANSWER: 3, AUTHORITY: 0, ADDITIONAL: 1

;; OPT PSEUDOSECTION:
; EDNS: version: 0, flags: do; udp: 4096
;; QUESTION SECTION:
;edu.           IN   DNSKEY

;; ANSWER SECTION:
edu.   86400   IN   DNSKEY  256 {ZSK of .edu}
edu.   86400   IN   DNSKEY  257 {KSK of .edu}
edu.   86400   IN   RRSIG   DNSKEY {signature on DNSKEY records}
...
```

In this response, the `.edu` name server has returned its public ZSK, public KSK, and a `RRSIG` type record over the two `DNSKEY` type records. We can use the public KSK to verify the signature on the public ZSK.

Because we trust the `.edu` name server's KSK (from the previous step), and the `.edu` KSK signs its ZSK, we now trust the `.edu` name server's ZSK.

Next, we query the `.edu` name server for the IP address of `eecs.berkeley.edu`.

```
$ dig +norecurse +dnssec eecs.berkeley.edu @192.5.6.30

;; Got answer:
;; ->>HEADER<<- opcode: QUERY, status: NOERROR, id: 60799
;; flags: qr; QUERY: 1, ANSWER: 0, AUTHORITY: 5, ADDITIONAL: 5

;; OPT PSEUDOSECTION:
; EDNS: version: 0, flags: do; udp: 4096
;; QUESTION SECTION:
;eecs.berkeley.edu.          IN  A

;; AUTHORITY SECTION:
berkeley.edu.        172800  IN  NS     adns1.berkeley.edu.
berkeley.edu.        172800  IN  NS     adns2.berkeley.edu.
```

```
berkeley.edu.        172800  IN  NS     adns3.berkeley.edu.
berkeley.edu.        86400   IN  DS     {hash of berkeley.edu's KSK}
berkeley.edu.        86400   IN  RRSIG  DS {signature on DS record}

;; ADDITIONAL SECTION:
adns1.berkeley.edu.  172800  IN  A      128.32.136.3
adns2.berkeley.edu.  172800  IN  A      128.32.136.14
adns3.berkeley.edu.  172800  IN  A      192.107.102.142
...
```

In this response, the `.edu` name server returns `NS` and `A` type records that tell us what name server to query next, just like in regular DNS.

In addition, the response has a `DS` type record and an `RRSIG` signature on the `DS` record. Sanity check: which key is used to sign the `DS` record?[2] Together, these two records sign the KSK of the `berkeley.edu` name server. Because we trust the `.edu` name server's ZSK (from the previous step), and the `.edu` ZSK signs the `berkeley.edu` KSK, we now trust the `berkeley.edu` name server's KSK.

Next, we query the `berkeley.edu` name server for its public keys.

```
$ dig +norecurse +dnssec DNSKEY berkeley.edu @128.32.136.3

;; Got answer:
;; ->>HEADER<<- opcode: QUERY, status: NOERROR, id: 4169
;; flags: qr aa; QUERY: 1, ANSWER: 5, AUTHORITY: 0, ADDITIONAL: 1

;; OPT PSEUDOSECTION:
; EDNS: version: 0, flags: do; udp: 1220
;; QUESTION SECTION:
;berkeley.edu.          IN  DNSKEY

;; ANSWER SECTION:
berkeley.edu.  172800  IN  DNSKEY  256 {ZSK of berkeley.edu}
berkeley.edu.  172800  IN  DNSKEY  257 {KSK of berkeley.edu}
berkeley.edu.  172800  IN  RRSIG   DNSKEY {signature on DNSKEY records}
...
```

In this response, the `berkeley.edu` name server has returned its public ZSK, public KSK, and a `RRSIG` type record over the two `DNSKEY` type records. We can use the public KSK to verify the signature on the public ZSK.

Because we trust the `berkeley.edu` name server's KSK (from the previous step), and the `berkeley.edu` KSK signs its ZSK, we now trust the `berkeley.edu` name server's ZSK.

Finally, we query the `berkeley.edu` name server for the IP address of `eecs.berkeley.edu`.

```
$ dig +norecurse +dnssec eecs.berkeley.edu @128.32.136.3

;; Got answer:
;; ->>HEADER<<- opcode: QUERY, status: NOERROR, id: 21205
;; flags: qr aa; QUERY: 1, ANSWER: 2, AUTHORITY: 0, ADDITIONAL: 1

;; OPT PSEUDOSECTION:
; EDNS: version: 0, flags: do; udp: 1220
;; QUESTION SECTION:
;eecs.berkeley.edu.          IN  A
```

[2] A: The ZSK of the `.edu` name server.

```
;; ANSWER SECTION:
eecs.berkeley.edu.  86400   IN  A      23.185.0.1
eecs.berkeley.edu.  86400   IN  RRSIG  A {signature on A record}
```

This response has the final answer `A` type record and a signature on the final answer. Because we trust the `berkeley.edu` name server's ZSK (from the previous part), we also trust the final answer.

33.7. Nonexistent domains

Remember that DNS is designed to be fast and lightweight. However, public-key cryptography is slow, because it requires math. As a result, name servers that support DNSSEC sign records *offline*–records are signed ahead of time, and the signatures saved in the server along with the records. When the server receives a DNS query, it can immediately return the saved signature without computing it.

Offline signing works fine for existing domains, but what if we receive a request for a nonexistent domain? There are infinitely many nonexistent domains, so we cannot sign them all offline. However, we cannot sign requests for nonexistent domains *online* either, because this is too slow. Also, online cryptography makes name servers vulnerable to an attack. Sanity check: what's the attack?[3]

DNSSEC has a clever solution to this problem–instead of signing individual nonexistent domains, name servers pre-compute signatures on *ranges* of nonexistent domains. Suppose we have a website with three subdomains:

```
b.example.com
l.example.com
q.example.com
```

If we sort every possible subdomain alphabetically, there are three ranges of nonexistent domains: everything between `b` and `l`, `l` and `q`, and `q` and `b` (wrapping around from z to a).

Now, if someone queries for `c.example.com`, instead of signing a message proving the nonexistence of that specific domain, the name server returns a **NSEC record** saying, "No domains exist between `b.example.com` and `l.example.com`. Signed, name server."

NSEC records have a slight vulnerability - notice that every time we query for a nonexistent domain, we can discover two valid domains that we might have otherwise not known. By traversing the alphabet, an attacker can now learn the names of every subdomain of the website:

1. Query `c.example.com`. Receive NSEC saying nothing exists between `b` and `l`. Attacker now knows `b` and `l` exist.
2. Query `m.example.com`. Receive NSEC saying nothing exists between `l` and `q`. Attacker now knows `q` exists.
3. Query `r.example.com`. Receive NSEC saying nothing exists between `q` and `b`. Attacker has already seen `b`, so they know they have walked the entire alphabet successfully.

Some argue that this is not really a vulnerability, because hiding a domain name like `admin.example.com` is relying on security through obscurity. Nevertheless, an attempt to fix this was implemented as **NSEC3**, which simply uses the hashes of every domain name instead of the actual domain name.

```
372fbe338b9f3bb6f857352bc4c6a49721d6066f (l.example.com)
6898bc7daf3054daae05e8763153ee1506e809d5 (q.example.com)
f96a6ec2fb6efbe43002f4cbf124f90879424d79 (b.example.com)
```

[3] A: Denial of service (DoS). Flood the name server with requests for nonexistent domains, and it will be forced to sign all of them.

The order of the domain names has changed, but the process is the same - if someone queries for `c.example.com`, which hashes to `8dca64e4b6e1724f0d84c5c25c9354d5529ab0a2`, the NSEC3 record will say, "No domains exist that hash to values between `6898b...` and `f96a6...`. Signed, name server."

Of course, an attacker could buy a GPU and precompute hashes to learn domain names anyway... and NSEC5 was born. Fortunately, it's still out of scope for this class.

34. Denial-of-Service (DoS)

34. Denial-of-Service (DoS)

34. Denial-of-Service (DoS)

34.1. Introduction

Since the bandwidth in a network is finite, the number of connections a web server can maintain is limited. Each connection to a server needs some minimum amount of network capacity in order to function properly. When a server has used up its bandwidth (the ability of its processors to respond to requests), any additional attempted connections are dropped. Any attack that is designed to cause a machine or a piece of software to be unavailable and unable to perform its basic functionality is known as a denial of service (DoS) attack.

A majority of DoS attacks refer to deliberate attempts to exceed the maximum available bandwidth of a server usually through exploiting program flaws or resource exhaustion. The underlying concept is that since different parts of the system might have different resource limits, the attack only needs to exhaust the part of the system with the least resources (i.e. the bottleneck). Since attackers in a DoS attack are not concerned with receiving responses from the server, they often spoof the source IP address in an attempt to obscure the identity of the attacker and make mitigations of the attack more difficult. Because some servers may stop DoS attacks by dropping all packets from certain blacklisted IP addresses, attackers can generate a unique source IP address for every packet sent, thus preventing the target from successfully identifying and blocking the attacker. This use of IP spoofing therefore makes it more difficult to target the source of a DoS attack.

34.2. Application Level DoS

Application level DoS attacks tend to target the resources that an application uses and exploits features of the application itself. Some attacks rely on asymmetry wherein a small amount of input from the attack results in a large amount of consumed resources. Such attacks could include exhausting the filesystem space by having continuous calls to write, exhausting the RAM by having continuous calls to malloc, exhausting the processing threads by having continuous calls to fork, or exhausting the disk I/O operations. Defense against such attacks usually take on a three-pronged approach:

1. Identification: You must be able to distinguish requests from different users and require some method to identify or authenticate them (though this process might be expensive and itself vulnerable to DoS attacks)
2. Isolation: You must ensure that one user's actions do not affect another user's experience
3. Quotas: You must ensure that users can only access a certain proportion of resources. There are many possible implementations of this. One method to implement this is to place specific limits on each user such as limiting users to only 4 GB of RAM and 2 CPU cores. Another example of is to assign specific roles to users such that only trusted people can execute expensive requests. Another possible "defense" would include proof-of-work (like CAPTCHA) wherein you force users to spend some resources in order to issue a request. The idea here is that the DoS attack becomes more expensive for the attacker as they have to now spend extra resources in order to succeed.

34.3. SYN Flood Attacks

Recall (from Chapter 31) that in order to initiate a TCP session, the client first sends a SYN packet to the server, in response to which the server replies with a SYN/ACK packet. This handshake is concluded with

the client sending a concluding ACK packet to the server, but if the server does not receive the ACK packet, it waits for a certain time-out period before discarding the session.

In a SYN flooding attack, the attacker sends a large number of SYN packets to the server, ignores the SYN/ACK replies, and never sends the ACK response. In fact, an attacker will usually use a spoofed IP source address in the SYN packets, so any SYN/ACK replies are sent to random IP addresses. Therefore, if the attacker sends a large number of SYN packets with no corresponding ACK packets, the server's memory will fill up with sequence numbers that it is forced to remember in order to match up TCP sessions with the expected ACK packets. Since these ACK packets never actually arrive, this wasted memory will ultimately block out other, legitimate TCP session requests.

Essentially, if the attacker sends a large volume of SYN packets to the server, the server is forced to send SYN/ACK packets back to the "client" and has to remember the sequence numbers of each of the packets for when the connections are established. However, if the attacker does not complete the handshake by sending the ACK packet, the server has wasted a lot of memory by being forced to remember all the sequence numbers for connections that will never actually happen, thus using up all of the server's bandwidth and preventing legitimate connections from taking place.

There are a couple of possible defenses for SYN flooding. The first is a process known as overprovisioning wherein we ensure that the server has a lot of memory. However, this can be pretty expensive and usually can still be circumvented depending on your threat model. Perhaps a more stable defense is to filter packets to ensure that only legitimate connections will create state, through the use of SYN cookies. In an ideal scenario, the server generates state for the client but does not save it when it sends the SYN/ACK flag; instead, it sends the state to the client encoded with a secret. It is then up to the client to store the state on behalf of the server and return the state in the corresponding ACK packet. Only when the handshake is complete will the server allocate state for the connection after checking the cookie against the secret. The issue, however, is that TCP does not have the mechanism to store state. Thus, instead, the server generates state for the client when it generates the SYN/ACK flag and does not save it; instead, it encodes the state within the sequence number with a secret. The client remembers the sequence number and returns it in the corresponding ACK number. Only when the handshake is complete will the server allocate state for the connection after checking the cookie against the secret.

Essentially, what is happening here is that the server does not create state until the handshake is completed, so the attacker cannot spoof source addresses.

34.4. Distributed Denial of Service (DDoS)

Today, most standard DoS attacks are impractical to execute from a single machine. Modern server technology allows websites to handle an enormous amount of bandwidth, much greater than the bandwidth that is possible from a single machine. However, DoS conditions can still be created by using multiple attacking machines in what is known as a Distributed Denial of Service (DDoS) attack. Here, malicious user(s) leverage the power of many machines (the number of machines could be in the thousands) to direct traffic against a single website in an attempt to create DoS conditions (i.e. prevent availability). Often, attackers carry out DDoS attacks by using botnets, a series of large networks of machines that have been compromised and are controllable remotely.

Theoretically, there is no way to completely eliminate the possibility of a DDoS attack since the bandwidth that a server is able to provide its users is always going to be limited. However, measures can still be taken to mitigate the risks of DDoS attacks. For example, several servers incorporate DDoS protection mechanisms that analyze incoming traffic and drop packets from sources that are consuming too much bandwidth. Unfortunately, IP spoofing makes this defense extremely difficult by obscuring the identity of the attacker bots and providing inconsistent information on where network traffic is coming from.

35. Firewalls

35. Firewalls

35.1. Introduction to Controlling Network Access

Suppose you are given a machine and asked to harden it against external attacks. How would you go about doing it?

A possible starting point might be to look at the functionality and the network services that this machine is providing to the outside world. If any of the network services have bugs or security flaws, an attacker could exploit that part of the service and might be able to penetrate your machine. As we know, bugs are inevitable, and bugs in security-critical applications often lead to security holes. Therefore, the more network services that your machine runs, the greater the risk of attacks.

The general principle here is that bugs present in code that you do not run cannot hurt you. Therefore, the less functionality you try to provide, the less of an opportunity exists for security vulnerabilities in that functionality. This suggests one simple way to reduce the risk of external attack: *Turn off every unnecessary network service*. By disabling every network-accessible application that isn't absolutely needed, you are essentially building a stripped-down box that runs the least amount of code necessary. After all, any code that you don't run, can't hurt you. And for any network service that you do run, double-check that it has been implemented and configured securely, and take every precaution possible to ensure that it is safe.

While this is an intuitive and fairly effective approach when you only have one or two machines to secure, the problem becomes slightly more complicated when we scale things up. Suppose you are now in charge of security for all of Caltopia, and your job is to protect the computer systems, the network and its infrastructure from external attacks. If Caltopia has thousands of computers, it will be extremely difficult to harden every single machine individually as each computer could have different operating systems and different hardware platforms. Furthermore, different users could have very different requirements, where a service that could be disabled for one user might be essential to another user's job. At this scale, it is often hard just to get an accurate list of all machines inside the company, and if you miss even one machine, it could then become a vulnerable point that could be broken into and serve as a launching point for attackers to attack the rest of the Caltopia network. So, managing each computer individually is probably infeasible at this scale.

However, it is still true that one risk factor is the number of network services that are accessible to outsiders. This suggests a possible defense; if we could block, *in the network*, outsiders from being able to interact with many of the network services from running on internal machines, we could potentially reduce the risk. This is exactly the concept behind *firewalls*, which are devices designed to block access to network services that are running on internal machines. Essentially, firewalls reduce risk by blocking network outsiders from having unwanted access to all the network services by acting as a choke point between the internet (outsiders) and your internal network. Now, all we need to know to implement a firewall is:

1. What is our *security policy*? Namely, which network services should be made visible to the outside world and which should be blocked? How do we discern insiders from outsiders?
2. How will we *enforce the security policy*? How do we build a firewall that does what we want it to do and what are the implementation issues?

35.2. Security Policy

If you wanted to visualize the topology of the internal network, you could think about having an internal network, which hosts all of the company's machines, the external world, which is the rest of the internet, and a communications link between the two.

How do we decide which computers have to be affected by the firewall and which don't? A very simple threat model could have us decide that we trust all company employees, but we don't trust anyone else. Thus, we define the internal network to contain machines owned by trusted employees and the external world to include everyone (and everything) else. The link to our Internet Service Provider (ISP) could be the link between the two networks.

Perhaps the simplest security policy that we could easily implement would be an outbound-only policy. Before we delve into how it works, let's define and distinguish between inbound and outbound connections. An *inbound connection* is one that is initiated by external users and attempts to connect to services that are running on internal machines. On the other hand, an *outbound connection* is one which is initiated by an internal user, and attempts to initiate contact with external services. An outbound-only connection would permit all outbound connections, but inbound connections would be denied outright. The reasoning behind such a connection is that internal users are trusted, so if they wish to open a connection, we will let them. The effect of the resulting connection is that none of our network services are visible to the outside world, but they can still be accessed by internal users. The issue is that such a security policy is likely too restrictive for any large organization since it means that the company cannot run any public web server, a mail server, an FTP server, etc. Therefore, we need a little more flexibility in defining our security policy.

More generally, our security policy is going to be some form of *access control policy*, wherein we have two subjects: a generic internal user and an anonymous external user. We then define an *object* to be the set of network services that are run on all inside machines; if there are 100 machines, and each machine runs 7 network services, then we have 700 objects. An access control policy should specify, for each subject and each object, whether that subject has permission to access that object.

A firewall is used to enforce a specific kind of access control policy, one where insider users are permitted to connect to any network service desired, whereas external users are restricted; there are some services that are intended to be externally visible (and thus, external users are permitted to connect to those services), but there are also other services that are not intended to be accessible from the outside world (and these services are blocked by the access policy).

As a security administrator, your first job would be to identify a security policy, namely which services should external users have access to and which services should external users not have access to. Generally, there are two main philosophies that we might use to determine which services we allow external users to connect to:

- *Default-allow or blacklist*: By default, every network service is permitted unless it has been specifically listed as denied. Under this approach, one might start by allowing external users to access all internal services, and then mark a couple of services that are known to be unsafe and therefore should be blocked. For example, if you learn today that there is a new threat that targets Extensible Messaging and Presence Protocol (XMPP) servers, you might be inclined to revise your security policy by denying outsiders access to your XMPP servers.
- *Default-deny or whitelist*: By default, every network service is denied to external users, unless it has been specifically listed as allowed. Here, we might start off with a list of a few known servers that need to be visible to the outside world (which have been judged to be reasonably safe). External users will be denied access to any service that is not on this list of allowed services. For example, if Caltopia users complain that their department's File Transfer Protocol (FTP) server is inaccessible to the outside world (since it is not on the allowed list), we can check to see if they are running a safe and properly configured implementation of the FTP service, and then add the FTP service to the "allowed" list.

The default-allow policy is more convenient since, from a functionality point of view, everything stays working. However, from a security point of view, the default-allow policy is dangerous since it *fails-open*, meaning that if any mistake is made (if there is some service that is vulnerable but you forgot to add it to the "deny" list),

the result is likely to be some form of an expensive security failure.

On the other hand, the default-deny policy *fails-closed*, meaning that if any mistake is made (if some service that is safe has been mistakenly omitted from the "allow" list), then the result is the loss of functionality or availability, but not a security breach. When operating at large scales, such errors of omission are likely to be common. Since errors of omission are a lot more dangerous in a default-allow policy than in a default-deny policy, and because the cost of a security failure is often a lot more than the cost of a loss of functionality, default-deny is usually a much safer bet.

Another advantage of the default-deny policy is that when a system fails open (like in default-allow), you might never notice the failure, and since attackers who penetrate the system are unlikely to tell you that they have done so, security breaches may go undetected for long periods of time. This gets you into an arms race, wherein you have to keep up with all of the attacks that adversaries discover, and even stay ahead of them. This arms race is usually a losing proposition since there are a lot more of the attackers than there are defenders and the attacker only has to win once to make you extremely vulnerable. In contrast, when a system fails closed (like in default-deny), someone will probably notice the flaw and will likely complain that some service is not working; since the omission is immediately evident and easily correctable, failures in default-allow systems are much more costly than failures in default-deny systems.

As such, a majority of well-implemented firewalls implement a default-deny system, wherein the security policy specifies a list of "allowed" services that external users are permitted to connect to, and all other services are forbidden. To determine whether a service should be removed from the allowed list, some kind of risk assessment and cost-benefit analysis is applied; if some service is too risky compared to its benefits, it is removed from the allowed list.

To identify network services, recall that TCP and UDP connections can be uniquely identified by the machine's IP address and port number. Therefore, we can identify each network service with a triplet (m, r, p), where m is the IP address of a machine, r is the protocol identifier (i.e. TCP or UDP), and p is the port number. For instance, the company might have its official web server hosted on machine 1.2.3.4, and then (1.2.3.4, TCP, 80) would be added to the allowed list. In a default-deny policy, the list of network services that should be externally visible would be represented as a set of these triplets.

35.3. Enforcement: Stateful Packet Filters

The main idea behind enforcing security policies is to do so at a choke point in the network. The existence of a central choke point gives us a single place to monitor, where we can easily enforce a security policy on thousands of machines with minimal effort. This idea of a choke point is similar to that of physical security; at an airport, for example, all passengers are funneled through a security checkpoint where access can be controlled. It is easier to perform such checks at one, or a few, checkpoints rather than hundreds or even thousands of entrances.

A stateful packet filter is a router that checks each packet against the provided access control policy. If the policy allows the packet, it is forwarded on towards its destination; if the policy denies the packet, then the packet is dropped and is not forwarded. The access control policy is usually specified as a list of rules; as the firewall processes each packet, it examines the rules one-by-one, and the first matching rule determines how the packet will be handled.

Typically, rules specify which connections are allowed. The rule can list the protocol (tcp or udp), the initiator's IP address (the machine that initiated the connection), the initiator's port number, the recipient's IP address (the machine that the connection is directed to), and the recipient's port number. A rule can use wildcards, denoted by the symbol $*$, for any of these. Each rule also specifies what action to take for matching connections; typical values might be ALLOW or DROP.

For example, take the following ruleset:
allow tcp $* : * \rightarrow$ \$ 1.2.3.4:25\$
drop $*$ $* : * \rightarrow * : *$

This ruleset allows anyone to open a TCP connection to port 25 on machine 1.2.3.4, but blocks all other connections.

A stateful packet filter maintains state, meaning that it keeps track of all open connections that have been established. When a packet is processed, the filter allows the firewall to check whether the packet is part of a connection that is already open. If it is, then the packet can be forwarded. Without state, it is harder to know how to handle the packet; for example, if we see a packet that is going from X to Y, we don't know if the packet was on a connection that was initiated by X or by Y, the answer to which might determine whether or not the packet is allowed to be forwarded. By keeping state, stateful packet filters allow policies that inspect the data, like for example, a policy that blocks any attempt to log into an FTP server with the username "root". However, stateful packet filters must be written extremely carefully to ensure that it only keeps a small amount of information per connection to ensure that the firewall does not run out of memory.

35.4. Enforcement: Other Firewalls

Stateless packet filters tend to operate on the network level and generally only look at TCP, UDP, and IP headers. In contrast to stateful packet filters, stateless packet filters do not keep any state, meaning that each packet is handled as it arrives, with no memory or history retained by the firewall.

Application-layer firewalls restrict traffic according to the content of the data fields. These types of firewalls have certain security advantages since they can enforce more restrictive security policies and can transform data on the fly.

Rather than simply inspecting traffic, we can also build firewalls that participate in application layer exchanges. For example, we can introduce a web proxy in a network and configure all local systems to use it for their web access. The local web browsers would then connect to the proxy rather than directly to remote web servers, and the proxy would in turn make the actual remote request. A major benefit of this design is that we can include monitoring in the proxy that has available for its decision-making all of the application-layer information associated with a given request and reply, so we can make fine-grained allow/deny decisions based on a wealth of information. This sort of design isn't specific to web proxies but can be done for many different types of applications. The general term is an application proxy or gateway proxy. One difficulty with using this approach, however, is implementation complexity. The application proxy needs to understand all of the details of the application protocol that it mediates. Another potential issue concerns performance. If we bottleneck all of the site's outbound traffic through just a few proxy systems, they may be overwhelmed by the load.

35.5. Firewall Principles

In general, the mechanism that enforces an access control policy often takes the form of a reference monitor. The purpose of a reference monitor is to examine every request to access any controlled resource (an "object") and determine whether that request should be allowed.

There are three security properties that any reference monitor should have:

- Unbypassable (also known as Always invoked): The reference monitor should be invoked on every operation that is controlled by the access control policy. There must be no way to bypass the reference monitor (i.e., the complete mediation property): all security-relevant operations must be mediated by the reference monitor.
- Tamper-resistant: The reference monitor should be protected from tampering by other agents. For instance, other parties should not be able to modify its code or state. The integrity of the reference monitor must be maintained.
- Verifiable: It should be possible to verify the correctness of the reference monitor, including that it actually does enforce the desired access control policy correctly. This usually requires that the reference monitor be extremely simple, as generally it is beyond the state of the art to verify the correctness of subsystems with any significant degree of complexity.

We can recognize a firewall as an instance of a reference monitor. How are these three properties achieved?

- Always invoked: We assumed that the packet filter is placed on a chokepoint link, with the property that all communications between the internal and external networks must traverse this link. Thus, the packet filter has an opportunity to inspect all such packets. Moreover, packets are not forwarded across this link unless the packet filter inspects them and forwards them (there needs to be no other mechanism by which packets might flow across this link). Of course, in some cases we discover that it doesn't work out like we hoped. For instance, maybe a user hooks up an unsecured wireless access point to their internal machine. Then anyone who drives by with a wireless-enabled laptop effectively gains access to the internal network, bypassing the packet filter. This illustrates that, to use a firewall safely, we'd better be sure that our firewalls cover all of the links between the internal network and the external world. We term this set of links as the security perimeter.
- Tamper-resistant: We haven't really discussed how to make packet filters resistant to attack. However, they obviously should be hardened as much as possible, because they are a single point of failure. Fortunately, their desired functionality is relatively simple, so we should have a reasonable chance at protecting them from outside attack. For instance, they might not need to run a standard operating system, any user-level programs, or network services, eliminating many avenues of outside attack. More generally, we can use firewall protection for the firewall itself, and not allow any management access to the firewall device except from specific trusted machines. Of course, we must also ensure the physical security of the packet filter device.
- Verifiable: In current practice, unfortunately the correctness of a firewall's operation is generally not verified in any systematic fashion. The software is usually too complex for this to be feasible. And we do suffer as a result of our failure to verify packet filters: over time, there have been bugs that allowed attackers to defeat the intended security policy by sending unexpected packets that the packet filter doesn't handle quite the way it should. In addition, experience has shown that firewall policies rapidly become complex. Thus, even if a firewall's internal workings are entirely correct, the rules it enforces may not in fact accurately reflect the access controls that the operator believes they provide.

Finally, firewalls also embody *orthogonal security* meaning that it can be deployed to protect pre-existing legacy systems much more easily than other security mechanisms that have to be integrated with the rest of the system. A reference monitor that filters the set of requests, dropping unallowed requests but allowing allowed requests to pass through unchanged, is essentially transparent to the rest of the system: other components do not need to be aware of the presence of the reference monitor.

35.6. Firewall Advantages

- Central control: A firewall provides a single point of control. When security policies change, only the firewall has to be updated; we do not have to touch individual machines. For instance, when a new threat to an Internet service is discovered, it is often possible to very quickly block it by modifying the firewall's security policy slightly, and all internal machines benefit from this protection. This makes it easier to administer, control, and update the security policy for an entire organization.
- Easy to deploy: Because firewalls are essentially transparent to internal hosts, there is an easy migration path, and they are easy to deploy (incrementally, or all at once). Because one firewall can protect thousands of machines, they provide a huge amount of leverage.
- Solve an important problem: Firewalls address a burning problem. Security vulnerabilities in network services are rampant. In principle, a better response might be to clean up the quality of the code in our network services; but that is an enormous challenge, and firewalls are much cheaper.

35.7. Firewall Disadvantages

- Loss of functionality: The very essence of the firewalls concept involves turning off functionality, and often users miss the disabled functionality. Some applications don't work with firewalls. For instance, peer-to-peer networks have big problems: if both users are behind firewalls, then when one user tries to connect to another user, the second user's firewall will see this as an inbound connection and will usually block it. The observation underlying firewalls is that connectivity begets risk, and firewalls are all about managing risk by reducing connectivity from the outside world to internal machines. It should be no surprise that reducing network connectivity can reduce the usefulness of the network.

- The malicious insider problem: Firewalls make the assumption that insiders are trusted. This gives internal users the power to violate your security policy. Firewalls are usually used to establish a security perimeter between the inside and outside world. However, if a malicious party breaches that security perimeter in any way, or otherwise gains control of an inside machine, then the malicious party becomes trusted and can wreak havoc, because inside machines have unlimited power to attack other inside machines. For this reason, Bill Cheswick called firewalled networks a "crunchy outer coating, with a soft, chewy center." There is nothing that the firewall can do once a bad guy gets inside the security perimeter. We see this in practice. For example, laptops have become a serious problem. People take their laptop on a trip with them, connect to the Internet from their hotel room (without any firewall), get infected with malware, then bring their laptop home and connect it to their company's internal network, and the malware proceeds to infect other internal machines.
- Adversarial applications: The previous two properties can combine in a particularly problematic way. Suppose that an application developer realizes their protocol is going to be blocked by their users' firewalls. What do you think they are going to do? Often, what happens is that the application tunnels its traffic over HTTP (web, port 80) or SMTP (email, port 25). Many firewalls allow port 80 traffic, because the web is the "killer app" of the Internet, but now the firewall cannot distinguish between this application's traffic and real web traffic.

The fact that insiders are trusted has as a consequence that all applications that insiders execute will be trusted, too, and when such applications act in a way that subverts the security policy, the effectiveness of the firewall can be limited (even though the application developers probably do not think of themselves as malicious). The end result is that, over time, more and more traffic goes over ports nominally associated with other application protocols (particularly port 80, intended for web access), with firewalls gaining less and less visibility into the traffic that traverses them. As a result firewalls are becoming increasingly less effective.

36. Intrusion Detection

36. Intrusion Detection

In this class, we've talked about many ways to prevent attacks, but not all defenses are perfect, and attacks will often slip through our defenses. How do we detect these attacks when they happen?

Imagine that you're managing a local network of computers (for example, all the web servers and employee computers in a company's office building). The local network is connected to the Internet with a router (recall that all requests from the local network to the wider Internet will pass through this router). How can we detect attacks on this network?

36.1. Types of detectors

There are three broad types of detectors. The main difference in implementation is where on the network these detectors are installed. Each type of detector has its advantages and drawbacks.

36.2. Types of detectors: Network Intrusion Detection System (NIDS)

A NIDS (network intrusion detection system) is installed between the router and the internal network. This means that all requests to and from the outside Internet must pass through the NIDS. The NIDS can see (and potentially modify) all packets sent to the outside Internet and received from the outside Internet.

The biggest advantage of a NIDS is that a single NIDS is enough to cover the entire network. There's no need to install anything on the end hosts (e.g. employee computers or web servers) because all their requests will pass through the NIDS anyway. Installing a single NIDS for the whole network is a cheap solution with low management overhead.

However, there are some drawbacks to using a NIDS. Recall that even though the Internet fundamentally works by sending packets, rich information communicated with higher-layer protocols are made up of multiple packets. For example, a single message sent through TCP may consist of many small packets that are combined to form a longer message. Also, packets may be dropped or sent out of order–it's the end hosts' responsibility to rearrange the pieces correctly with TCP.

A plain NIDS that just observes individual packets would not be too useful, because it will probably see a lot of packets with partial data out of order. The NIDS may also be seeing packets from lots of different TCP connections, since every connection from inside the network goes through the NIDS. A more useful NIDS would separate packets by their connection and correctly reorder the packets within each connection together by TCP sequence number. Once the NIDS has successfully reconstructed the connection, it can read the rich information and analyze it for attacks.

To make matters worse, the TCP connection reconstructed at the NIDS may not match the TCP connection that the end host sees. Recall that each TCP packet has a time-to-live (TTL) field, which specifies how long the packet can be in transit before it expires. (This is often measured in the number of hops, i.e. the number of machines that the packet has been sent through.) Then there could be a scenario where the NIDS receives a TCP packet because the TTL has not yet expired, but by the time it's sent to the end host, the TTL has expired, and the end host discards the message. There could also be a scenario where the NIDS sees a packet,

but it gets corrupted or dropped before it reaches the recipient. Thus the NIDS must also reason about packets that potentially don't reach the end host.

The possibility of inconsistent interpretations of messages between the NIDS and the end host can be exploited for attacks. Consider a NIDS that raises an alert for an attack if it encounters the string `/etc/passwd` in any request. An attacker could send a packet with the content `%65%74%63/%70%61%73%73%77%64`. To a basic NIDS doing string matching, this won't look like an attack, but if the end host is expecting a URL-encoded string and decodes this string, then the end host will receive the string `/etc/passwd`. The NIDS has failed to detect a potential password attack! This type of attack, where the attacker tries to obfuscate the contents of an attack, is called an *evasion attack*. The possibility of evasion attacks suggests that not only does the the NIDS have to reason about inconsistent information about connections, but the NIDS must also reason about how the end hosts may potentially interpret the information in the connection.

Another major issue with NIDS is the need to deal with encrypted traffic. Most modern web traffic is encrypted with HTTPS (TLS), which is end-to-end secure. In other words, the NIDS has no way to determine the contents of the messages being sent. To allow NIDS to analyze encrypted traffic, the network may need to be configured so that the end hosts give the NIDS their private keys to allow the NIDS to decrypt TLS connections. This might not always be a desirable solution, since it compromises the security of private keys and the security guarantees of NIDS, and it may allow network analysts to see sensitive information that only the end hosts should see.

36.3. Types of detectors: Host-based Intrusion Detection System (HIDS)

A HIDS (host-based intrusion detection system) is installed directly on the end hosts. For example, antivirus software might be considered a HIDS, because it is installed on the same computer that is generating and receiving network requests.

HIDS have much fewer inconsistency issues than NIDS. Since the HIDS is located on the same machine that is receiving and interpreting the requests, it can directly check what data is received and how the data is being parsed. HTTPS connections are also no longer an issue, because the HIDS can view the decrypted traffic at the end host.

However, these advantages don't come for free. Unlike NIDS, where a single implementation can defend against the entire network, a HIDS must be installed for every machine on the network. This can be very costly, especially if different machines need differently-configured HIDS.

HIDS also don't defend against all evasion attacks. For example, a web server might expect a filename input from the user and serve the matching file to the user. If the user inputs `evanbot.txt`, the server might check the `/public/files` directory and return the `/public/files/evanbot.txt` file to the user. An attacker could supply a malicious input like `../../etc/passwd`. In Unix, `..` says to go up one directory, so this input would allow the attacker to access the passwords file, even though it's located in a different directory on the server. This type of attack is called a *path traversal attack*. To fully defend against path traversal attacks, it is not enough for the the HIDS to understand the contents of the end request. The HIDS would also need to reason about how the underlying filesystem interprets the contents of the end request. This can lead to further parsing inconsistencies and evasion attacks.

36.4. Types of detectors: Logging

A third approach to intrusion detection is logging. Most modern web servers generate logs with information such as what web requests have been made, what files have been accessed, and what applications have been run. We can analyze these logs for evidence of malicious behavior or attacks.

Logging is similar to HIDS because both systems directly use information from the end host, avoiding many potential parsing inconsistencies and problems with encrypted traffic. However, like NIDS, logging may need to consider evasion attacks such as the path traversal attack, which requires smarter filesystem parsing and can lead to inconsistencies.

The biggest drawback to logging is that it cannot be done in real-time. By the time the log has been generated, the event that's being logged has already happened. This also means that if an attack has happened, a log-based system will only detect the attack after it has happened. This can be dangerous if the attack is immediately damaging. However, logs are still useful for detecting attacks after they've happened (better late than never).

In terms of cost, logging is usually cheap, because web servers already have built-in logging mechanisms. The only overhead is occasionally running an external script on those logs to search for evidence of attacks.

36.5. False Positives and False Negatives

There are two ways a detector can go wrong. A *false negative* occurs when an attack happens but the detector incorrectly reports that there is no attack. A *false positive* occurs when there is no attack, but the detector incorrectly reports that there is an attack. As an example, consider a fire alarm system. A false negative occurs if there is a fire but the fire alarm does not go off. A false positive occurs if there is no fire, but the fire alarm goes off.

It's easy to build a detector with a 0% false negative rate. Just report that there is an attack every single time. Then there will never be a case where your detector incorrectly reports that there is no attack. Similarly, a detector that never reports an attack will have a 0% false positive rate. Clearly, both of these are pretty useless detectors. In the real world, different detectors will have different false negative rates and false positive rates, and part of designing a good detector is balancing the two error rates. In general, as one error rate decreases, the other error rate will increase. Intuitively, to get fewer false positives, you must alert less often, which means you will also incorrectly fail to alert more often (higher false negative rate). Similarly, to get fewer false negatives, you must alert more often, which means you will also incorrectly alert more often (higher false positive rate).

Suppose you have two detectors. Detector A has a false positive rate of 0.1% and a false negative rate of 2%. Meanwhile, Detector B has a false positive rate of 2% and a false negative rate of 0.1%. Which of these detectors is better? It depends on the cost of each type of error. Consider the fire alarm system–if the fire alarm gives you a false negative, then your building has burned down, but if the fire alarm gives you a false positive, then you've wasted an hour with the fire department. In this scenario, the false positive is probably less costly than the false negative, so you would probably prefer Detector B. In another scenario, a false positive might be more costly than a false negative, so you might prefer Detector A instead.

The quality of your detector also depends on the rate of attacks. Consider Detector A again. If we receive 1,000 requests a day and 5 of them are attacks, then the expected number of false positives is $0.1\% \times 995 \approx 1$ request per day. (995 requests are not attacks, and out of the non-attacks, 0.1% of them will incorrectly be reported as an attack.) However, now suppose we receive 10,000,000 (10 million) requests a day and 5 of them are attacks. Now the expected number of false positives is $0.1\% \times 9{,}999{,}995 \approx 10{,}000$ requests per day. Note that nothing has changed about the detector. The only thing that changed was the number of requests received per day (and thus the rate of attacks). However, in the second scenario, our detector is much less useful, because we have to handle 10,000 false positives every day. This example shows that accurate detection is very challenging when the rate of attacks is extremely low, because even a very good detector will flag so many false positives that it becomes impractical to manually review every single false positive. For more information on this phenomenon, read about the base rate fallacy.

36.6. Detection strategies

So far, we've talked about how detectors are installed and how to measure their effectiveness, but we haven't talked about how the detector actually analyzes network traffic to detect an attack. There are four main strategies for detecting an attack, each with their benefits and drawbacks.

36.7. Detection strategies: Signature-based detection

The idea: Look for activity that matches the structure of a known attack.

Signature-based detection can be thought of as *blacklisting*–we maintain a list of patterns that are not allowed, and we detect if we see something in the list of disallowed patterns.

Example: We know that inputting some garbage bytes, followed by a memory address, followed by shellcode is the structure of a buffer overflow attack, so the detector can search for strings that match this pattern and classify them as attacks.

Pros:

- Detecting known signatures is easy.
- It's very good at detecting known attacks. Over time, the security community has built up huge shared libraries of attacks with known signatures.

Cons:

- It won't catch new attacks without known signatures.
- It might not catch variants of known attacks if the variant is different enough that the signature no longer matches. If the signature detector is too simple, it's easy to modify the attack slightly to circumvent the detector.

36.8. Detection strategies: Anomaly-based detection

The idea: Develop a model of what normal activity looks like. Flag any activity that deviates from normal activity.

Anomaly-based detection can be thought of *whitelisting*–we maintain a list of normal patterns that are allowed, and we detect if we see something that is *not* in the list of allowed patterns.

Example: A C program expects user input. Most user input consists of letters, numbers, and symbols–things you would expect a user to type on a keyboard. We determine that normal activity is any input that can be typed on a keyboard, and flag any input that cannot be typed on a keyboard. If an attacker tries to input a buffer overflow attack with memory addresses and shellcode (raw bytes that often can't be typed on a keyboard), we detect that this doesn't match normal behavior and flag it as an attack.

Pros:

- It can catch new attacks that have never been seen before.

Cons:

- Defining normal behavior is difficult. What if you train a model for normal behavior on training data that includes attacks?
- A poor model might classify lots of attacks as normal, or classify lots of normal requests as attacks.

In general, anomaly-based behavior is mostly studied in academic papers but not widely deployed as a detection strategy.

36.9. Detection strategies: Specification-based detection

The idea: Manually specify what normal activity looks like. Flag any activity that deviates from normal activity.

Specification-based detection is also a form of whitelisting. The main difference between specification-based detection and anomaly-based detection is that specification-based detection manually defines normal activity (instead of trying to learn a model for normal activity).

Example: A C programmer writes a program that asks for the user's age as input. The programmer knows that ages are numerical and specifies that normal behavior is inputting a number. If an attacker tries to input a buffer overflow attack with memory addresses and shellcode (raw bytes that are not numbers), we detect that this doesn't match normal behavior and flag it as an attack.

Pros:

- It can catch new attacks that have never been seen before.
- If the specification is well-defined, the false positive rate can be made very low.

Cons:

- It's very time-consuming to manually write specifications for every application.

36.10: Detection strategies: Behavioral detection

The idea: Look for evidence of compromise.

Unlike the other three models, behavioral detection doesn't search for attack patterns in the input. Instead, behavior detection looks for malicious behavior that an attacker might try to perform. In other words, we are looking for the result of the exploit, not the contents of the exploit itself.

Example: A C programmer writes a program that never calls the `exec` function. If an attacker tries to input a buffer overflow attack with shellcode that calls the `exec` function to spawn a shell, we detect that the code has called `exec` and flag this behavior as an attack. Note that we did not analyze the attacker input. Instead, we analyzed the program behavior and noticed that it called the `exec` function, which is evidence that the program has been compromised.

Pros:

- It can catch new attacks that have never been seen before.
- If the behavior rarely or never occurs in benign (non-attack) circumstances, the false positive rate can be made very low.
- It can be cheap to implement.

Cons:

- The attack is only detected after it's started, so there's no way to prevent the attack before it happens.
- An attacker can try to avoid detection by using different behavior to execute their attack.

37. Abusing Intrusion Detection

37. Abusing Intrusion Detection

On a high level, network intrusion detection can be thought of as wiretapping on a bulk scale. The NSA utilizes various "off-the-shelf" concepts including using various Network Intrusion Detection Systems and Databases, malicious code, and hadoop.

The NSA language is slightly different from the security language present in this class.

- A *selector* in NSA parlance is a piece of information that identifies what you are looking for, like an email address, a phone number, etc.
- A *fingerprint* in NSA parlance is an intrusion detection match
- An *implant* is a malcode or another piece of sabotage

The FISA (Foreign Intelligence Surveillance Act) Amendments Act section 702 states that if you are not a "US person", meaning you are not a US citizen or permanent resident, and you are located outside of the United States, then the NSA can obtain all your information through a US provider. If you are either a US person or are located within the United States, however, you are afforded a lot of protection due to the United States Constitution

The NSA is part of Five Eyes (FVEY), an intelligence alliance comprising of Australia, Canada, New Zealand, the United Kingdom, and the United States. These countries are parties to the multilateral UKUSA agreement, a treaty for joint cooperation in signals intelligence. The primary rule within FVEY is "when in country X, behave according to country X's laws".

The NSA's objective is, for a valid target (that is a non-US person outside of the US), to be able to collect all relevant communications. This, however, requires the capability to collect information on everyone since a valid target could be anyone, meaning that the NSA requires global capabilities. As such, the solution that the NSA employs is to collect all intelligence that they feasibly can on everybody and store it for as long as possible, assuming that at some point in the future they might need to search that information and hopefully find something useful. One issue of utilizing the aforementioned method, however, is that there is now too much information, and sifting through that information to find the relevant details is much more difficult.

Say, for example, that you are an analyst and you are watching an IRC chat between two "anonymous" people, and your task is to identify the two people involved in the conversation; the only information that you do have is that they both visited some article at some specified time. The first thing you might do is to use Signal Intelligence Flow (also known as the Digital Network Intelligence Flow) to develop an online pattern of life for these anonymous users, before using a computer network exploitation to invoke an "exploit by name" attack to take over their computers.

A majority of signals intelligence starts off with wiretaps, and the NSA's preferred system of doing so is called Xkeyscore Deepdive (a large majority, if not all, of which are overseas). These wiretaps are nothing more than scalable network intrusion detection systems (NIDS)! After the NIDS extracts the network information and parses the data packet to extract the metadata, it stores it within a dataframe. However, unlike conventional NIDS, if you want to evade the NSA monitoring, all you have to do is encrypt the data using some form of cryptography. In practice, Xkeyscore is primarily centered around an easy-to-use web interface with a lot of pre-canned search scripts for low-sophistication users along with a large number of pre-made "fingerprints" to identify applications, usages, etc.

Good transport cryptography causes significant problems when the NSA attempts to collect data; however, they are able to utilize some tricks to get around this (though a large majority of these don't work anymore). The wiretaps collect encrypted traffic and pass it off to a black-box elsewhere, usually at some datacenter. The NSA might come back at some point in the future having obtained the cryptographic key and might be able to then convert the ciphertext back into plaintext.

38. Malware

38. Malware

38.1. Overview

Malware, or malicious software (also known as malcode), is any type of attacker code that runs on victim computers. One of the primary ways that malware is able to propagate is through self-replicating code, which is a code snippet that outputs a copy of itself (usually to send to other people). For example, suppose a piece of malware runs on your computer; in addition to, say deleting all your files or turning on your webcam, the malware also outputs a copy of itself and sends it to other computers, thereby infecting other devices.

Viruses and worms are two categories of self-propagating malware wherein the malicious code sends copies of itself to other users. A virus is a piece of malware or malcode that requires some user action to propagate, meaning that the user has to take some action in order for the virus to spread. Usually, once the computer gets infected, a piece of code is stored somewhere on the infected computer. Then, when the user runs the code, the virus gets spread to other users.

On the other hand, a worm is a piece of malware or malcode that does not require user action to propagate. Usually, rather than the infection happening on code that is stored on the computer that gets run later, it instead infects a computer by altering some already-running code. As such, no user interaction is required for the worm to spread to other users.

One possible application of malware is to construct a botnet. A botnet is a set of compromised machines, or bots, that are under centralized control, allowing the owner of the botnet to own a huge amount of resources that could be used for other attacks (like DoS). An attacker could use a virus or a worm to infect a large number of machines, causing every infected machine to now be under the control of the attacker.

38.2. Viruses

Recall that viruses are forms of malware that require user action to propagate, meaning that it usually infects a computer by altering some stored code and when the user runs the code, the malicious code spreads to other users. For example, an attacker could infect the start-up code of an application, meaning that when the user tries to open the application, the malcode will run and look for opportunities to infect more systems (i.e. forward itself to other users, copy itself onto a USB drive, etc.).

One common approach to detecting viruses is through signature-based detection. Since viruses are self-propagating, they often use copies of the same code. Since signature-based detection uses patterns of known attacks, a signature can be created on the virus (since the virus has been infecting several computers in the same manner using the same code snippet). So, the signature-based detection system will capture a virus on one system (usually through a sacrificial computer which opens a bunch of malicious files on purpose) and look for bytes corresponding to the malcode on other systems. Antivirus software performs these checks for you by usually including a checklist of common viruses. Most antiviruses will incorporate some form of signature-based detection and will use the signatures of these viruses to ensure that your computer is not infected. Stronger antivirus softwares will likely have a greater number of virus signatures than weaker ones, ensuring that your computer is protected from a wider range of attacks.

Viruses have existed for several decades, and there is a constant race that exists between attackers writing viruses and antivirus companies detecting viruses. As this arms race continues, propagation strategies of

modern malware have evolved. Attackers tend to look for evasion strategies as they don't want to be detected by the antivirus software. As such, they could change the appearance of the virus so that each copy looks different, thus making signature-based detection much harder. Rather than changing the virus's appearance manually, certain evasion strategies attempt to automate this process through polymorphic code. In this arms race, since the attacker can see what detection strategies the antivirus software is using, but the antivirus cannot see what attacks the attacker is planning, the attackers often have a slight advantage. In other words, the attackers can see the defense strategies employed by the antivirus companies and therefore write evasion strategies to get around them (namely, the attacker knows the system). Therefore, since the detectors have to usually publish their code first, they are at a bit of a disadvantage.

38.3. Polymorphic and Metamorphic Code

In an attempt to continuously change the virus's appearance to avoid signature-based detection, attacks employ polymorphic code wherein each time the virus propagates, it inserts an encrypted copy of the malcode. This code also includes the key and the decryptor, so when the code runs, it uses the key and decryptor to obtain the original, plaintext malcode. Since encryption schemes produce different looking outputs on repeated encryptions (with IND-CPA secure schemes), the attacker is able to change the appearance of the virus to help avoid signature-based detections. However, note that encryption is being used for obfuscation and not for confidentiality. Namely, the attacker is not trying to hide the contents of the virus (rather, the malcode is going to get run eventually and the decoder and the key are sent in plaintext), but simply avoid detection by making every copy of the virus look different. As such, this also means that weaker encryption algorithms, like ECB, can be used (since our goal is not confidentiality) and the decryption keys can be sent in plaintext.

One possible defense against polymorphic code is to simply add a signature for detecting the decryptor code. For example, a possible signature could be a key being used to decrypt a certain piece of code. However, this raises a lot of false positives since there are a lot of pieces of code that are not malware, which use a key to decrypt other pieces of code. Furthermore, another issue arises if the decryptor code is scattered across different parts of memory as matching several small instructions is a lot harder than matching one big block of code. Another possible defense is to run the potentially dangerous code in a sandbox, or an isolated environment, where if something goes terribly wrong, nothing outside of the sandbox is affected. For example, if a piece of code performs a decryption mechanism, the machine could execute the code in a sandbox (like a VM), thus allowing us to analyze the code structure without actually executing the code in a dangerous environment.

In addition to polymorphic code, metamorphic code is another way to try to avoid signature-based detection. Here, each time the virus propagates, it generates a semantically different version of the code. In other words, the code performs the same high-level action, but with minor differences in execution, like changing variable names or changing the order of certain operations or using a for loop instead of a while loop. Usually included in metamorphic code is a code rewriter which changes the code randomly each time. Note that the rewriter can also change the rewriter code in addition to the virus code before propagating the virus to ensure that the entire malcode looks different.

Because the code is now changing, there is now no easy pattern to find the malcode, meaning that signature-based detection is extremely difficult. However, it does let us use behavioral-detection instead, wherein we analyze the behavior of the code instead of the syntax (since the syntax is continuously changing). As such, we now look at the effect of the instructions rather than the appearance of the instructions. However, viruses can subvert behavioral detection; for example, the virus could delay analysis by waiting a long time before executing the malicious code or it could detect that the code is being analyzed (run in a debugger or a sandbox) and could choose different, "normal" behavior.

Theoretically however, it is pretty much impossible to write a perfect algorithm to separate malicious code from safe code (though if you do manage to write something that accomplishes this task, you would have solved the halting problem!). Rather, antivirus softwares usually try to simply look for new and unfamiliar code. The software company keeps a central repository of previously-seen code and if some code has never been seen before, it treats that piece of code as malicious. Flagging unfamiliar code is a powerful defense as

it employs a signature-based detection system to detect malicious behavior as well as a strategy for people avoiding the first detector. In other words, if the attacker does not modify the code for each propagation, it will have a detectable signature and if the attacker modifies the code each time, it always appears as new, and therefore suspicious.

38.4. Worms

Worms are pieces of code that, unlike viruses, do not require user action to propagate; instead, they usually infect a computer by altering some already-running code. Since worms want to run immediately, they usually randomly choose machines by randomly generating 32-bit IP addresses and try connecting to them in an attempt to propagate. Essentially, worms want to directly inject malcode into a lot of different computers very quickly. To find the different computers to inject, the worm will either try to connect to random machines or will use a pre-generated "hit-list".

Worms can potentially spread extremely quickly since they parallelize the process of propagation and replication. As more computers are infected, more computers are available to spread the worm further. While viruses have the same property, they usually spread more slowly since user action is needed to activate the virus. As such, worm propagation can be modeled as an infectious epidemic and computer scientists often use the same models that biologists use to model their spread of infectious diseases. Similar to epidemics, the spread of the worm depends on the size of the population, the proportion of the population that is vulnerable to the infection, the number of infected hosts, and the contact rate, or how often the infected host communicates with other hosts. The number of infected hosts grows logistically, meaning that the initial growth is exponential, since as more hosts are infected, there are more opportunities to infect, but later growth slows down as it becomes harder to find new non-infected hosts to infect.

39. Anonymity and Tor

39. Anonymity and Tor

39.1. Overview

Imagine you wanted to use a group messaging app to share a public message but did not want your name attached to your message. How would you go about using the chat to achieve that?

One possible solution would be to randomly choose one member of the group on the participant list, send them a private message with your message and have them post the message with their name instead. This way your message gets posted to the forum but without your name attached to it.

This is the basis of *anonymity*, essentially a methodology that enables you to conceal your identity. In the context of the Internet, we may want *anonymous communications* wherein the identity of the source and/or destination are concealed. Note that anonymity is not the same as confidentiality, which is about keeping the message private, whereas anonymity is concerned with keeping one's identity private. As we will soon come to understand, anonymity on the internet is difficult without help; baked into the internet protocol is placing the IP address into the packet header, meaning that any message that is sent to a server reveals your IP address (which reveals quite a bit about your identity and is often one of the primary means of internet tracing). Malicious users, however, have it easier when it comes to anonymity due to the existence of *botnets*, which come about when someone hacks a set of machines and then controls those machines for various purposes (each machine is essentially a bot in the hands of the person who hacked it). The key takeaway here is that for the good guys to gain anonymity, we generally require some help, and what this comes down to is to simply ask someone else to send the message for you (as we saw in the previous example).

A majority of anonymity techniques require the use of a *proxy*, an intermediary (usually a somewhat trusted 3rd party) who relays our traffic for us. Say for example you wanted to visit a website, *www.example.com*, but you don't want to reveal to that server that you are visiting them. To hide your identity, you can route your traffic through a proxy server, which visits *www.example.com* on your behalf, gets the page, and returns the page to you. An issue that you might be thinking about right now is that you have to place your trust into this proxy server as they are now able to have access to your identity, so you are *not quite* completely anonymous, and you would be completely correct. We will see a couple of ways to circumvent this further down the road.

39.2. Anonymity in Cryptography

Let's try to understand in a little more detail how anonymity works when dealing with cryptography. Say that Alice wants to send a message M to Bob, but wants to conceal her identity when doing so. This means that not only should Bob not know that M is from Alice, but an evesdropper, Eve, should also not be able to determine that Alice is communicating with Bob. Using an intermediate proxy server, Alice can encrypt (M, Bob) using the proxy server's public key (K_{PS}), and sends this along to the intermediary. Note that the information packet needs to include the message and the person who the intermediary has to forward the information along to, in this case, Bob. Since Alice wants to send this information privately, she utilizes public key encryption. The proxy then decrypts the packet from Alice, retrieves the message, M, and the intended recipient, Bob, and then forwards the message to the intended destination. Essentially, all the proxy server is doing is accepting encrypted messages, decrypting them to extract the message and the destination, and then forwarding the message to the destination.

Notice that this method, as mentioned earlier, requires Alice to have complete trust in the proxy server in the hopes that they will not reveal her identity. To prevent having to place full trust in a third party server, we use the concept of onion routing.

39.3. Onion Routing

The key idea behind onion routing is the use of multiple proxy servers chained together in the hopes that at least one can be trusted. Again, Alice wants to send the message M to Bob, but this time she doesn't quite trust her proxy server, so instead decides to use onion routing to chain together 3 different intermediaries (let's call them Frank, Dan, and Charlie).

- Charlie is going to be the final proxy in the chain, so it needs to know the final recipient and the final message. So the packet Charlie receives will be (M, Bob) encrypted with Charlie's PK, $K_{Charlie}$ since Alice wants the message to be private. Charlie then decrypts the packet with his private key to obtain (M, Bob), then sends the message M to Bob.
- Since Alice doesn't trust Charlie to not reveal her identity, however, she wants to route the message through another proxy server, Dan. Again, the proxy requires the message and the recipient it forwards the message to. In this case, the message is $(M, Bob)_{KCharlie}$, and the recipient is Charlie. So the packet that Dan receives is $((M, Bob)_{KCharlie}, Charlie)_{KDan}$. Dan decrypts the packet using his private key to obtain $((M, Bob)_{KCharlie}, Charlie)$, then sends $M' = (M, Bob)_{KCharlie}$ to Charlie.
- Since Alice doesn't trust Dan either to not reveal her identity, she again wants to route the message through a different proxy server, Frank. Again, the proxy requires the message and the recipient that it forwards the message to. In this case the message is $((M, Bob)_{KCharlie}, Charlie)_{KDan}$ and the recipient is Dan. So the packet that Frank receives is $(((M, Bob)_{KCharlie}, Charlie)_{KDan}, Dan)_{KFrank}$. Frank decrypts the packet using his private key to obtain $(((M, Bob)_{KCharlie}, Charlie)_{KDan}, Dan)$, then sends $M'' = ((M, Bob)_{KCharlie}, Charlie)_{KDan}$ to Dan.

The overall routing scheme sends the packet from $Alice \rightarrow Frank \rightarrow Dan \rightarrow Charlie \rightarrow Bob$, where Frank, Dan, and Charlie are the three intermediaries, or proxies:

- Alice sends Frank $(((M, Bob)_{KCharlie}, Charlie)_{KDan}, Dan)_{KFrank}$
- Frank decrypts this using his private key and sends $((M, Bob)_{KCharlie}, Charlie)_{KDan}$ to Dan
- Dan decrypts this using his private key and sends $(M, Bob)_{KCharlie}$ to Charlie
- Charlie decrypts this using his private key and sends M to Bob
- For additional security, the message M could also have been encrypted using PK_{Bob}.

Note that this approach can be generalized to n intermediaries, and no one proxy knows both the sender and the recipient. In fact, even if $n - 1$ of the intermediaries were malicious and were colluding, as long as one recipient is honest, there is a low probability that they can connect Alice and Bob. If there were not a lot of people using the system however, if Frank and Charlie colluded, there would be a way to link the messages. In reality, though, each proxy server would likely be sending and receiving several thousands of messages, so it would be extremely difficult to link these two messages together.

Tor uses this type of onion routing for anonymous web browsing and censorship circumvention. While this example only depicted onion routing in one-direction, it can easily be scaled up, and Tor provides bidirectional communication.

39.4. Onion Routing Issues and Attacks

One of the downsides in onion routing is performance, as the message has to bounce off of several proxy servers before it reaches its destination, it takes a lot longer to get there. Each time the message goes through a proxy, there is an extra delay that is added to the latency. However, it should be noted that performance decreases linearly with the number of proxies added, and if you estimate that roughly half of the available proxy servers are honest then you only need to chain together a small number of proxies before you can be fairly sure that you have gained some level of anonymity (you can think of security going up exponentially, but performance going down linearly with the number of proxy servers added).

Another possible attack is one that was mentioned in the previous section, when the first and last proxy servers are under malicious control. The attacker can use timing information to link Alice and Bob, but, as mentioned earlier, this depends on the amount of traffic that is flowing through the proxy servers at that time. A possible defense is to pad messages (this is what Tor does but they note that it's not enough for defense), introduce significant delays, or if Bob is ready to accept encrypted messages, then the original message M can be encrypted.