5/10/2024

CMSC414 FINAL EXAM Review

DISCLAIMER: THIS MATERIAL IS MADE BY A STUDENT AND MAY CONTAIN ERRORS. CONSUME WITH A “GRAIN OF SALT” AND PLEASE CONSULT DAVE OR A TA WITH ANY QUESTIONS

RISHABH BARAL

AUTHOR'S NOTE TO READERS

Dear Classmates,

As we conclude our journey through CMSC414: Network Security under the guidance of Professor Dave Levin, I wanted to take a moment to express my gratitude for the enriching academic experience we've shared. Throughout the course, we've delved into a plethora of topics that have not only expanded our understanding of network security but also equipped us with invaluable knowledge for our future endeavors in the field.

From the intricacies of Stack Layout and Buffer Overflow to the complexities of Command Injection and Malware, each topic has served as a building block in our comprehension of the multifaceted nature of network security. Our discussions on SQL Injection, XSS/CSRF vulnerabilities, and the significance of ML Security have shed light on the evolving landscape of cyber threats and the indispensable role of proactive defense mechanisms.

Moreover, our exploration of Symmetric Key Crypto, TLS, PKI, and the consequences of Cryptography misuse has underscored the importance of robust encryption protocols in safeguarding sensitive information. The elucidation of Onion Routers, Networking protocols, and IP Security/VPNs has provided us with a comprehensive understanding of the mechanisms employed to ensure confidentiality, integrity, and availability in network communications.

Furthermore, our examination of TCP Attacks, DoS mitigation strategies, and the implementation of DNSSEC has deepened our appreciation for the intricate interplay between security measures and network performance. Through rigorous analysis and hands-on exercises, we've not only grasped the theoretical foundations but also honed our practical skills in identifying and mitigating security vulnerabilities.

As we part ways and embark on our respective journeys, let us carry forth the knowledge and insights garnered from CMSC414: Network Security. Let us remain vigilant in our pursuit of fortifying digital infrastructures and fostering a safer cyber ecosystem for generations to come.

Thank you, Professor Levin, for your unwavering dedication and guidance throughout this course. Your expertise and passion have been instrumental in shaping our understanding of network security and inspiring us to strive for excellence in this dynamic field. Wishing you all continued success and fulfillment in your future endeavors.

Warm regards,

Rishabh Baral

Contents

Table of Figures

Stack Layout and Memory Refresher

Before we can even begin discussing how to make programs, and in turn our networks, more secure we must understand how a program is laid out in memory. In order to do this, let's have a quick refresher on the stack layout, which we will model based on how it appears on the Linux operating System, although it shouldn't be much different for other operating systems. Linux models its stack using what is called the "process model" in which the stack layout is defined one process at a time.

Recall that all programs are stored in memory. Now let's assume we have a 4GB memory (a little small, but it makes for easier explanation). In the "process model" of memory management, **a process thinks that it owns ALL the available memory.** Remember those strings of hex digits you got when you ran GDB? These are addresses, but they have no reference point. They are **virtual** addresses that are mapped to actual addresses by the OS or CPU. When storing data in memory, the location in which data is stored depends on how it is created.

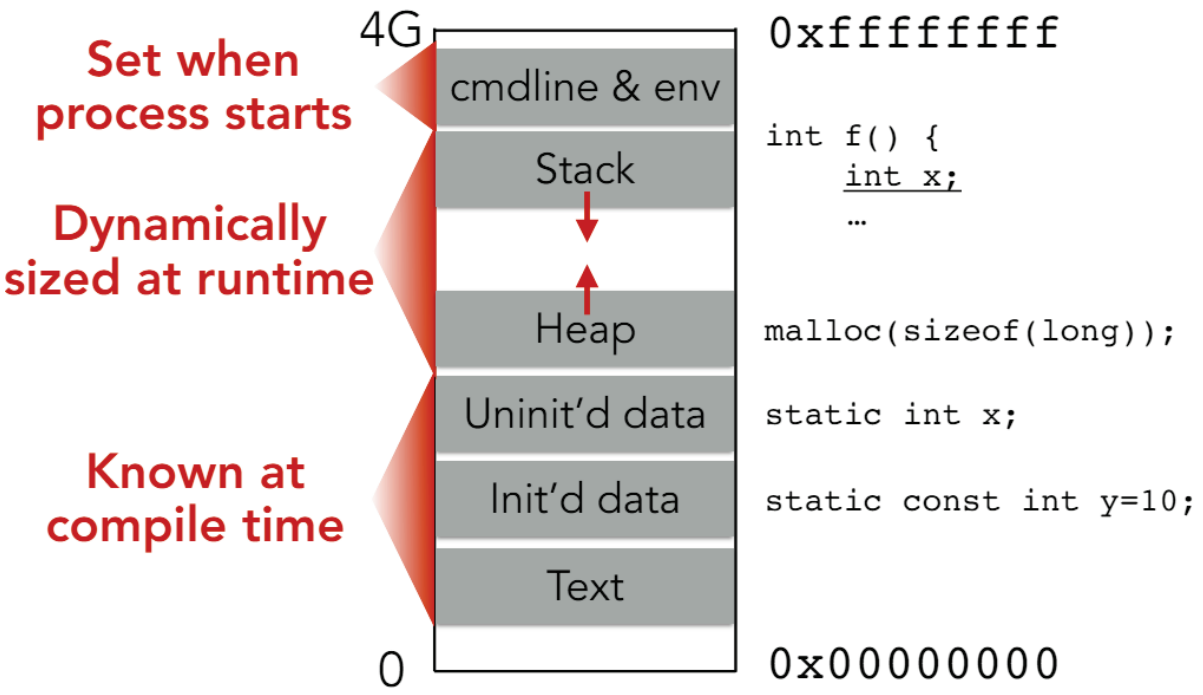


Figure 1: The stack layout of a process in memory

Now that we have a basic knowledge of how memory looks when we create a program, we can wreak havoc by exploiting this very memory layout in a series of runtime attacks. Notice from the memory diagram above that the stack and heap grow in opposite directions. Furthermore, remember that the compiler actually **gives instructions to dynamically change the size of the stack at runtime**. For the sake of simplicity, and also because it's where most attacks propagate from, let's just look at the stack for now.

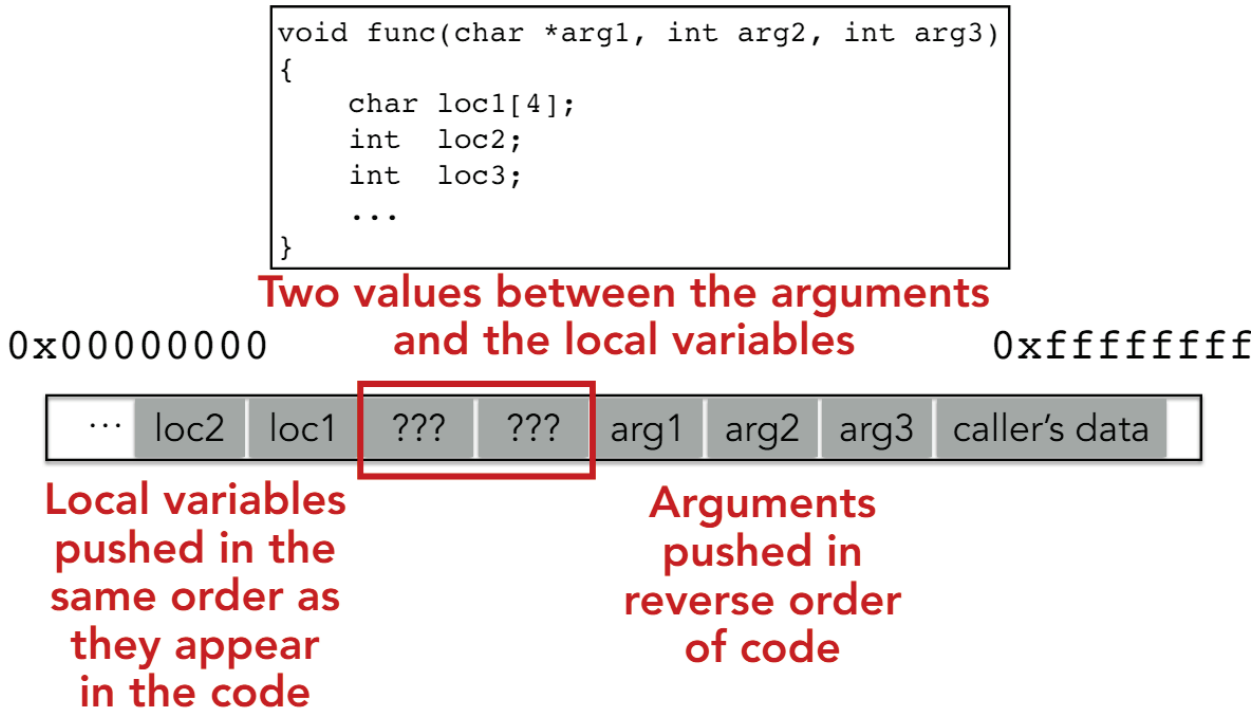


Figure 2: Stack Layout showing pre-processing in a function

In order to better understand the diagram above and also in order to fill in the two spots marked ???, let's imagine this function if it were to run. Let's say we are performing some operation on the second local variable, loc2. We have no idea where in memory this function is apportioned, and we also have no idea how many arguments there are (could be a variable amount). However, we know that each little "box" in the figure is 4 bytes, meaning that the variable will always be 8 bytes before the block marked ???. This is where we can set what is called the frame pointer. In order to find where we were, we need a virtual "bookmark". This is done by assigning another pointer called the stack pointer to the first of two blocks marked ??? in which the frame pointer is then stored. Now, this raises another issue. If we do continue executing and there is a call to a second function somewhere later down the line, we want to be able to resume where we left off. This is done by filling in the second block marked ??? with what is known as the instruction pointer.

So just to summarize, when we create a function, there are a few steps that the OS takes to ensure that the function does not utterly break program execution. Before it begins executing, it will push the caller's data (return address), push the function arguments (in reverse), and then jump to the beginning of the function. While running, the function adds its own frame and instruction

pointers before pushing on its local variables and continuing execution. At the end of the function, the program jumps back to the previous stack frame, resets it, and leaves the address. As a final clean-up, the program then removes all arguments to the function from the call stack and pops the frame off the stack.

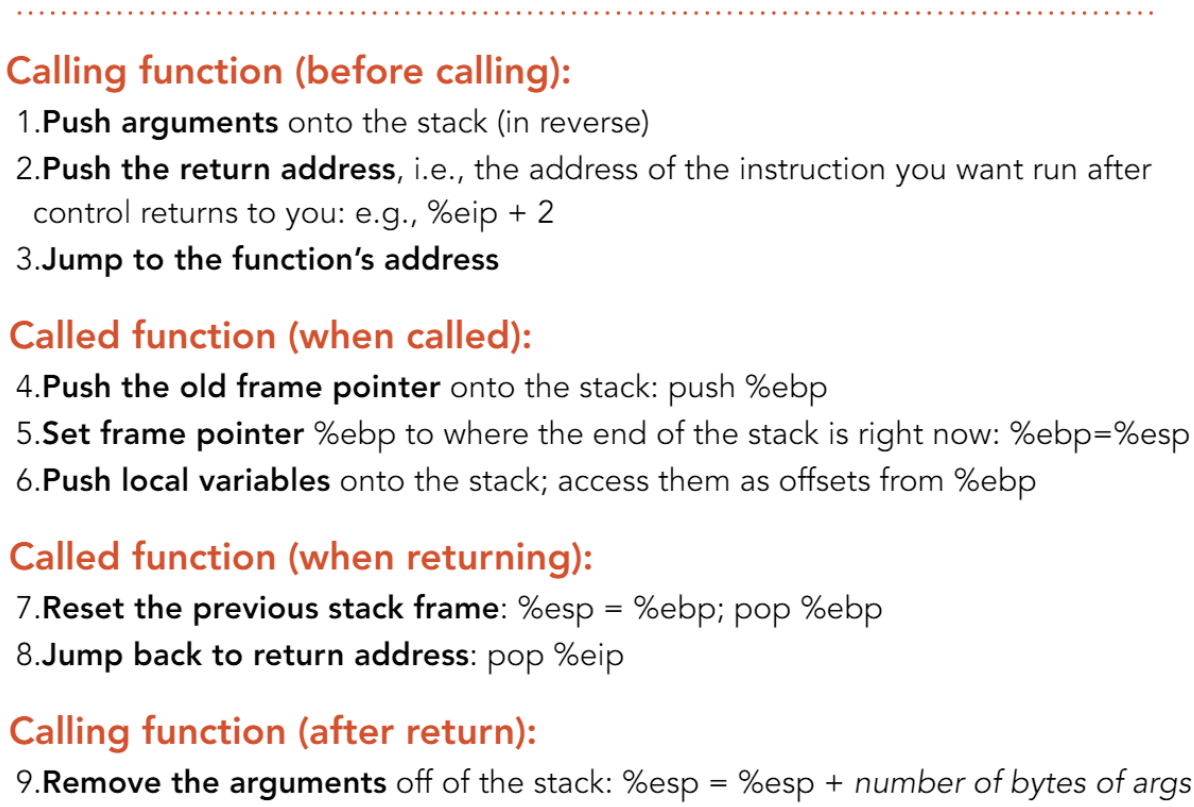


Figure 3: A Summary of the process undertaken in running a function (Linux Model)

Classic Memory Attacks and Defenses

Whenever memory attacks come to mind, all programming students sort of groan because we have all at least once conducted an unintended memory attack. If you remember those dreaded **SEGFAULTs** we used to get when coding in C, those are an example of buffer overflow which is the easiest and most common memory attack. Recall that strings in C are null terminated (end with /0). Recall from Figure 2 that the local variables are stored in memory right next to the base pointer (%ebp) and the instruction pointer (%eip). This can become problematic if a locally stored buffer ends up overwriting these two variables with data. This can happen if functions such as strcpy/strncpy are used, but only in a case where the destination's size is **smaller** than the text being copied. For example, let's assume that we have a buffer of size 4 for college name (we'll store UMD), but we accidentally copy 10 characters (Univ of MD) into it. In such a scenario, if the buffer was in local variable 2 (loc2), what happens is that the memory looks something like this (C allows us to keep writing until it hits a null byte):

U n i v	_ o f _	M D /0	
55 6e 69 76	20 6f 66 20	4d 44 00 00	00 00 00 00
Buffer	loc1	%ebp	%eip

Result: Program received signal SIGSEGV, Segmentation fault.

0x00005555555551cb in main ()

Figure 4: Rudimentary diagram of Stack layout showing Buffer Overflow resulting from copying string of size 7 into buffer of size 4

In doing so, what we have done is we have overflowed the original buffer by 6 characters. In addition, we have overwritten local variable 1 (loc1) which isn't dangerous, but we have also overwritten our base pointer with 0x0000444d, an address that likely cannot be accessed. Thus, when the operating system tries to retrieve the ebp and return to the address stored therein, it ends up trying to access memory it does not have permission for, resulting in that all dreaded SEGFAULT.

Now, buffer overflow on user-defined strings is fairly common. But because everything in memory, for simplicity's sake, is assumed to be adjacent, the problem of a simple buffer overflow can quickly spiral out of control. The logical progression from buffer overflow is to a class of attacks known as "code injection". In Code Injection attacks, buffer overflows are exploited to wreak havoc by executing user-defined code that has been stored in memory. Usually, this code is some sort of full-purpose shell with the intention of privilege escalation (guest/non-user to root). The key to defending this attack is to make the complicated things nearly impossible. Firstly, we need to figure out how to load the code itself. Recall that functions like strcpy, strncpy, and strlcpy stop reading if there is a null byte (byte of all zeroes), meaning that we must construct the code not to have any null bytes, not use the loader (it's injected code) and not use the stack (we're destroying the stack with the overflow). Now let's assume that we have written code to do what we want and were mindful of the restrictions placed on null bytes. The second challenge is actually getting the code to

run. We only have access to the memory from the buffer onwards, meaning that we need to be clever in how we manipulate it to make our code run. Recall that the next instruction upon return from a function is saved in the instruction pointer (%eip), which is stored 2 blocks from the local variables. So, in order to make sure our code runs, we need to hijack the saved instruction pointer by replacing it with the address at the beginning of our code. However, we cannot reliably know where in memory the code actually begins. One approach is to brute force the address since most operating systems run on either a 32-bit (or 64-bit) architecture, meaning that you would likely only have to compute 2^32 (or 2^64) values at most. However, the smart solution is to use what is known as a "nop sled", a string of "nop" codes which would slide an instruction pointer one byte at a time until it reaches an actual operation. This makes running shellcode trivial since the only requirement is that the guess made must be within the nop sled in order to ensure that the shell is reached by the instruction pointer. So, in short, to execute a buffer overflow attack that runs a full-purpose shell, the stack should be laid out as follows:

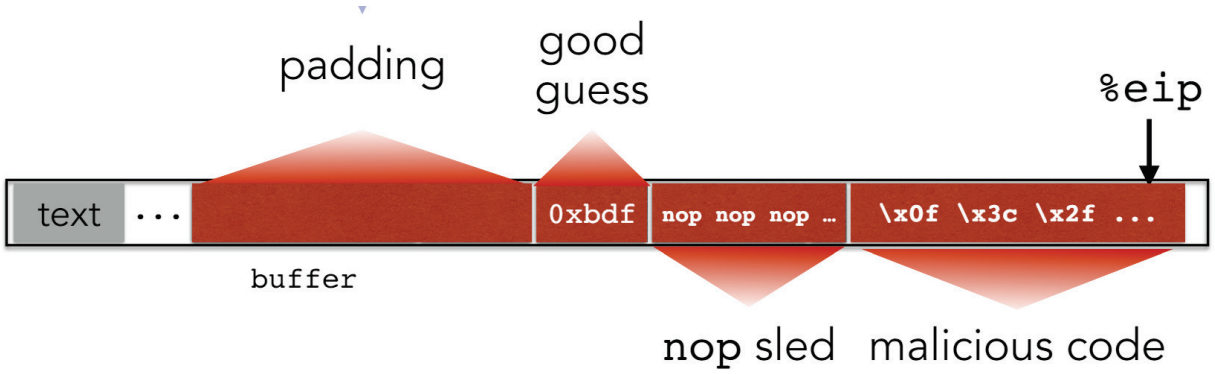


Figure 5: Stack Diagram showing layout for buffer overflow attack

Now that we have a rudimentary understanding of the buffer overflow attack, the next logical step is to understand possible methods of defense against such an attack. The first and simplest defense is to use a check-value known as a stack canary, by which a program will abort if the value of the canary is not what was expected. The second defense is to counter the second challenge. Recall that finding the return address is one of the trickiest parts of code injection. In order to defend against this, we can make what's already hard even harder. We do this in a process called ASLR (**A**ddress **S**pace **L**ayout **R**andomization) whereby the segments are in the same order but begin at different addresses from run to run. One of the attacks that was introduced with widespread use of ASLR was the return-to-libc attack, in which the location of usleep is used to hijack the pointer and run code injection. Furthermore, any debugger or forked process keeps the same offsets as the randomized original, which can make things easier through use of patterns. The last defense deals with the issue of hijacking the saved instruction pointer. In order to prevent this, the most common defense is to make the stack non-executable. However, this prevents any program from executing because most things need the stack to work. The second possible defense is to remove malicious libraries such as system() from libc, but this would crash the entire operating system. The last, and arguably most effective, defense against this is to use seccomp-bpf, a method in which a program is only allowed to access the functions in the c library that it needs for a complete and error-free execution.

Advanced Memory Attacks and Defenses

Now that we have looked at simple and easily executable memory based attacks and the common defenses deployed against them, we can now delve into the realm of more complex memory attacks and the defenses used to counteract them.

The first such attack is what is known as a Return-Oriented Programming, or ROP, attack. The way in which return-oriented programming attacks work is that they start at the end of a block (Return statement) and traverse backwards until valid instructions are found. For example, a function like ecb_crypt can have an ROP attack conducted against it. Just look at this snippet of code and see how different execution can be depending on where interpretation begins:

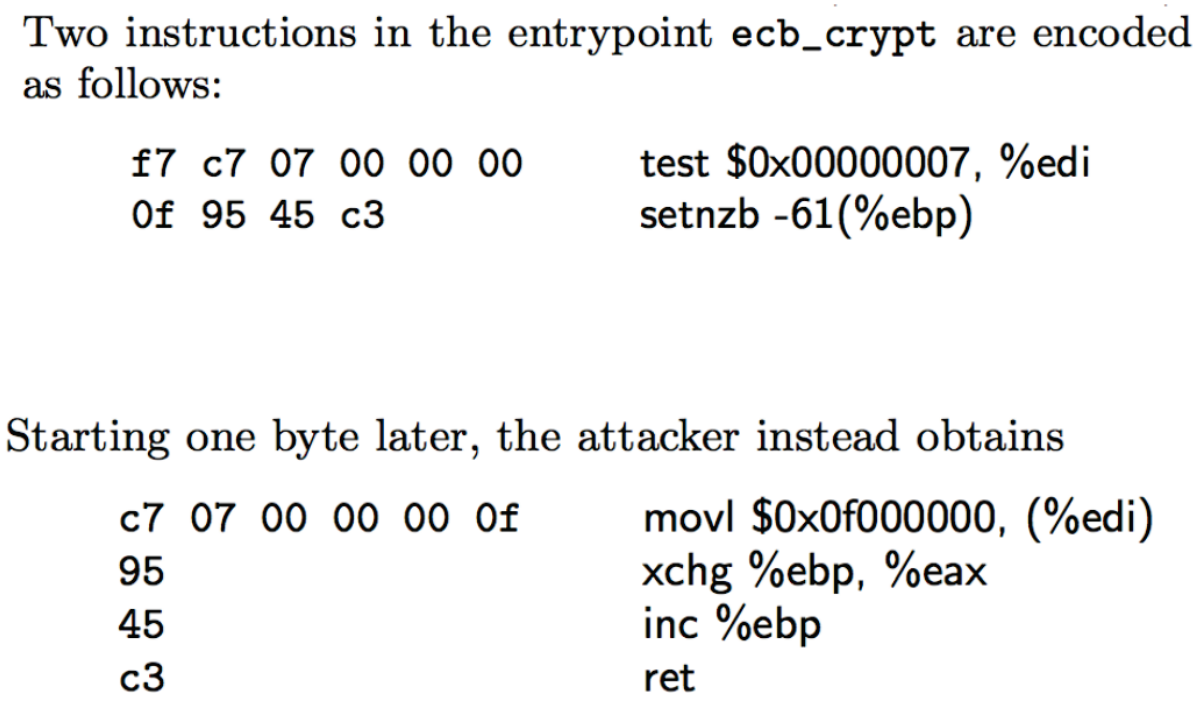


Figure 6: Code Example showing vulnerability to ROP attack in ecb_crypt function

In the frame of a ROP attack, each block (lines of code followed by a return) is called a **gadget**. These gadgets can be chained to wreak havoc on program execution if they are used in the right way. However, most gadgets do not involve the key registers, leading to harmless side effects if any.

The second class of advanced memory attacks returns to buffer overflow. They are called Format String Vulnerabilities, and they exploit the fact that functions like printf and fprintf print values based on format strings and user input. In addition, they use the fact that printf/fprintf/sprintf never check the number of arguments given and continue to print under the assumption that the correct number of arguments were provided. For example, using each of the following print statements has different side effects that a presumptive attacker could use to inject

and execute code.

- `printf("100% dml");`
  - Prints stack entry 4 byes above saved %eip
- `printf("%s");`
  - Prints bytes *pointed to* by that stack entry
- `printf("%d %d %d %d …");`
  - Prints a series of stack entries as integers
- `printf("%08x %08x %08x %08x …");`
  - Same, but nicely formatted hex
- `printf("100% no way!")`
  - ***WRITES*** the number 3 to address pointed to by stack entry

Figure 7: A listing of a few format string vulnerabilities when using printf in C

Another type of buffer overflow attack involves integers. If integers are used in code, and are not checked for validity, then the program could implode on itself in the event that the integer is used. In the figure below, there are two examples of integer overflow/underflow attacks. Each attack exploits the same vulnerability, a lack of a security check.

```
#define BUF_SIZE 16
char buf[BUF_SIZE];
void vulnerable()
{
    int len = read_int_from_network();
    char *p = read_string_from_network();
    if(len > BUF_SIZE) {
        printf("Too large\n");
        return;
    }
    memcpy(buf, p, len);
}
```

Negative

Ok

Implicit cast to unsigned

```
void vulnerable()
{
    size_t len;
    char *buf;
    len = read_int_from_network();
    buf = malloc(len + 5);
    read(fd, buf, len);
    ...
}
```

HUGE

Wrap-around

Figure 8:

(a) Showing an integer-based attack using negative numbers

(b) showing integer underflow attack based on memory allocation

Malware: Viruses

Malware is defined as malicious code that is stored on and runs on a victim's system. However, it doesn't run independently. Just as biological viruses need a host, so do computer viruses. So, with that analogy in mind, a virus doesn't run unless the virus is forced to run. There are a couple of methods by which a virus can be made to run. Firstly, an attacker could use a user-facing or network-facing service that happens to be vulnerable. Secondly, a developer could have inadvertently added a backdoor which the attacker exploits. Third, an attacker might disguise the virus as a social experiment, tricking the user into clicking or installing it. The two most dangerous methods however are the trojan horse virus (a good program with malware in it) and the drive-by download (a webpage installs the malware without the user realizing. Malware is dangerous in any form, but different malware has different functions based on the permissions it is encoded with. Malware could be as harmless as a simple brag ("I've taken control of your files!") or as dangerous as a rootkit (malware that hides from software based detection by modifying the kernel). Still other kinds of malware try to crash a machine by breeding "rabbits" in a process known as fork-bombing (which usually begins with some form of "while true"). Malware has different types, each of which run at different times. Some run on a timer (time-bomb), some run on a set of conditions (logic bomb), and still others run on attaching themselves to other pieces of code.

In order to classify viruses, we name them based on what they infect. Document Viruses infect documents of all types such as Word Documents, PDFs, and other formatted documents. Boot Sector Viruses work by hijacking the boot sector of a machine, running when the machine is booted as the machine will use the boot sector, which has been compromised. The last kind of viruses rely on certain portions of memory being used more frequently than others. Often, they will camp out in these portions of memory until they are called, and as such, these viruses are called Memory Resident Viruses.

Every virus functions in a different way and uses a different technology in order to attack its intended host. Just for a wide variety of cases, let's look at some famous case studies. First, we'll look at the Brain virus which impacted IBM PCs in 1997. The way it propagated was by copying itself into the boot sector of the PC and then intercepted read requests from floppy drives by posing as the boot sector itself. Let's look at an even more dangerous and undetectable virus next: the Sony XCP rootkit. This virus was found in Sony music CDs in 2005 and was used as an anti-copyright software to prevent users from accessing music without a Sony CD player. Arguably, however, the most dangerous virus was Stuxnet. Stuxnet was a virus that ran via a USB drive and exploited **four** zero-day exploits (exploits known only to the attacker until the attack). The main purpose of Stuxnet was to break nuclear centrifuges by slowly increasing their rotational frequency until they reached a dangerous threshold. It is understood that Stuxnet was deployed by the CIA and took out nearly 1000 of Iran's 5000 nuclear arms grade centrifuges.

Web Security: SQL Injection

Before we can discuss SQL injection, we need to understand Databases themselves. This is because at the heart of every query is a database lookup. Databases are made up of named tables that are defined by their columns (attributes) and rows (records). The unit of work on a database is what is called a **transaction**. Transactions, at least for the sake of security, should be ACID. ACID here is an acronym for **A**tomicity (transactions are either done fully or not done at all), **C**onsistency (The database is not made invalid by the execution of a given transaction), **I**solation (the resulting database following a transaction is not visible until the transaction has been completed), **D**urability (the completion of a transaction should be maintained in a database despite external adverse events). In order to query databases, there must be a standard query language defined. This is called SQL (**S**tructured **Q**uery **L**anguage). Typical SQL statements include keywords indicating transactions (SELECT, UPDATE, DROP, etc.), keywords indicating table information (FROM), and keywords indicating conditions to be met by the data being selected or changed (WHERE, LIKE, etc.). All SQL statements must be followed by a semicolon (;), but some may also be followed by an additional comment (indicated with " -- ").

Now that we have a basic knowledge of databases and how to query them, we can start to explore the security vulnerabilities of using databases. Let's take a hypothetical example. Let's say that we have a website (we built it for testing) where we have some login code in a PHP file. We also know that this code queries our database of users and logs in if the query returns **any rows at all**. This simple fact can be exploited if the login fields are not sanitized. For example, if the following username were entered: "frank' OR 1=1); -- ", then the website can be used without entering a password as the login query now becomes: "Select * from Users where (name = 'frank' OR 1=1); -- ". In this case, the 1=1 will always evaluate to true, so regardless of what the first half of the where clause returns, this query will **always** allow somebody to log in to the website. This exploitation of a database using SQL and conditional logic is just one of the many different types of attacks under the umbrella of **SQL Injection**.

Just as with any attack, there are countermeasures that can be taken. The simplest and most naïve approach to preventing SQL injection is to blacklist the problematic characters (', --, ;). However, this raises the issue of users with names containing those characters (O'Neil, O'Connor, etc.) which means you do sometimes want those characters. This makes it difficult to decide if/when the characters are bad. Another approach to preventing SQL Injection that does better than blacklisting but may still not be perfect is Whitelisting. In Whitelisting, the database operator checks that the user input is in some set of values known to be safe (an integer in the right range for example). Yet another approach to preventing SQL Injection that works better, but not perfectly, is escaping characters that could alter control ("\' , \; , \- , \\). However, this runs into the same issue as blacklisting since these characters may be needed at times. The strongest defense against SQL Injection, and one that works well if executed correctly, is to use prepared statements and bind the data later on. Essentially what this does is it forces the inputs to be of a certain type, all but eliminating the chance of SQL Injection (however the chance is still non-zero). Noting that there is no perfect solution to SQL Injection attacks, the best efforts can be made to mitigate the impact should such attacks take place. The best way to do so is to limit privileges (the commands/tables a user has access to) and to encrypt sensitive data stored anywhere in the database.

Web Security: Web Background

In interactions with web servers, everything circles around getting and putting resources which are identified by a URL. Typically, a URL will have three (sometimes four) parts: the protocol (HTTP, HTTPS, SFTP, FILE, etc.), the hostname or server (main URL), a path to a resource, and may **sometimes** include arguments if the resource in question requires them.

Typically, when browsing the web, most requests are issued using HTTP or HTTPS as the protocol. In inspecting the request headers sent from your browser to a website, often there will be certain fields that can be handy to know about. For example, the Referrer URL indicates the site from which the request **originated**. Upon receipt of a request, the site will then process that request and send back a response. The response contains many fields, but the main fields that are important are the field containing the **status code** and the field containing any **cookies** (state the website wants the browser to store on behalf of it).

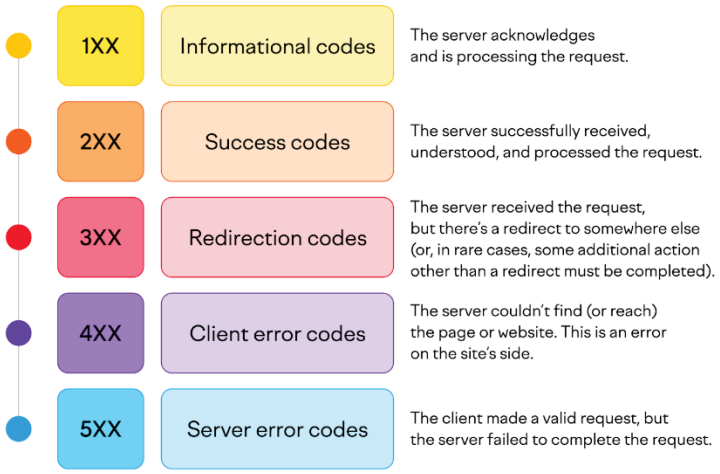


Figure 9: A simple diagram of common HTTP codes and their functions

Cookies

Most interactions with web servers happen over some form of transfer protocol (TCP, IP, HTTP, etc.). However, most of these protocols are stateless. If a client connects to a webpage, issues a request, receives a response, then continues this process for several minutes before disconnecting, before trying to reconnect later, the webpage has no identifying information to say that it is dealing with a client it has previously interacted with. In order to preserve state, the web server creates a cookie anytime a client makes an HTTP request to it. This cookie is then sent with the response and is used as an identifier in all future communication between the client and the server. However, cookies aren't used solely for preserving state. Cookies can also be used to save user preferences for personalization and may even be used for tracking.

Another common use of cookies is to keep track of users who have already authenticated themselves when visiting a website. If, for example, a user visited the website's login page with the correct information, then the server creates a "session cookie" with the user's information. Subsequent requests made during the session include the session cookie either in the headers or as one of the fields. The idea behind the use of a session cookie is to be able to identify browsers and users authenticated via the browser each time they log in to a website.

Attacking the Cookie Jar: XSS and CSRF

Typically, when sending GET requests to a server, they should not have a side effect, but some often do. Imagine a link of the form http://bank.com/transfer.cgi?amt=9999&to=attacker, if an active user visits the link, the side effect is that it will transfer $9999 from their bank account to an attacker. The idea behind CSRF (**C**ross-**S**ite **R**equest **F**orgery) is to use innocuous means of tricking the browser into visiting links with a side effect. This is most often done by disguising the malicious link as the source for an image file. The browser, none the wiser, visits the link to obtain what it believes is an image and ends up executing the side effect of the malicious link.

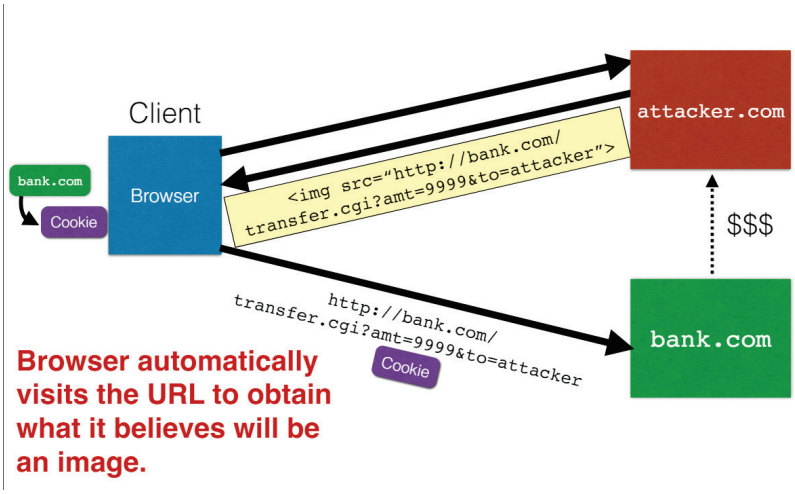


Figure 10: Simple Graphic showing an example of CSRF with a URL that has a side-effect

The main parts of Cross-Site Request Forgery are that there is an attack target (user with an account on a vulnerable website) and an attack goal (making requests to the server via the user's browser to disguise it as traffic coming from the user). Furthermore, the attacker may even have some tools, one of the key ones being disguising the request to the server (which has a predictable structure) as something that will get executed by default (source of an image).

The possibility of attack always raises the question of defense against such attacks. The naïve solution might be to disallow sites to link to one another, but the loss of functionality would be too high. The solution that is used in practice, however, is something called the Same Origin Policy. Under the Same Origin Policy, the browser associates webpage elements such as its layout and any associated cookies with a **given** origin. The Same Origin Policy ensures that only scripts received from **within** the origin of a webpage have access to that page's elements.

In order to subvert this policy, another kind of attack known as **Cross-S**ite **S**cripting (XSS) has arisen. In XSS, an attacker provides a malicious script which it tricks the browser into running by making it believe that the origin of the script is the website that the victim happens to be browsing at the moment. A general approach to XSS attacks is to trick the server of interest into sending the malicious script to the user. This will bypass the browser's same-origin policy because the origin is actually the same. Unlike CSRF which has only one type of attack, there are two types of XSS attack. Firstly, there is what is known as stored or persistent XSS whereby the attacker leaves their script on the server of interest, which later unwittingly sends it to your browser which then executes it as coming from the intended origin.

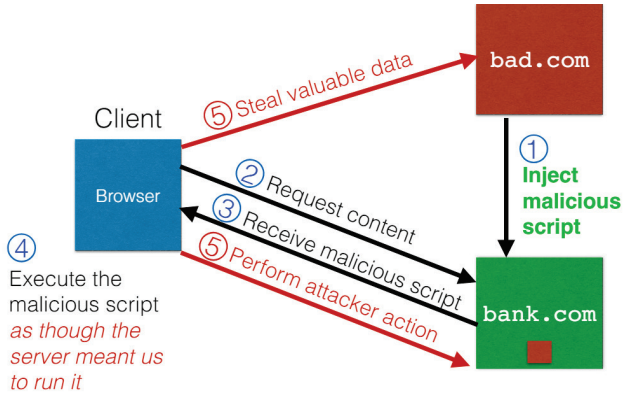


Figure 11: A simple but informative diagram of the process undertaken in a Stored XSS attack

The second type of XSS is called reflected XSS. A reflected XSS attack occurs when the attacker infects your machine with a script, which you send to the server of interest. The server of interest then echoes back the script in its response, and your browser executes it under the impression that the script was generated by the server.

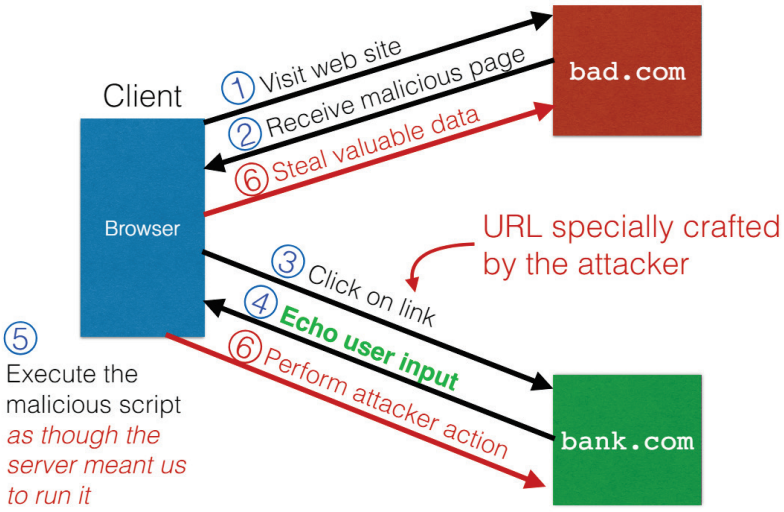


Figure 12: A simple but informative diagram showing the steps undertaken during a Reflected XSS attack

One common mistake in relation to XSS and CSRF is that the two are often confused for each other. XSS attacks exploit the trust that a **client browser** has in data sent from the **legitimate website** while CSRF attacks exploit the trust that a **legitimate website** has in data sent from the **client browser**.

Principles for Secure Design

When designing secure software, the naïve approach has always been to design and build the software *ignoring* security at first and building it in later. The improved, and agreed upon, approach is to build security into the software from the very beginning and incorporate security-minded thinking at each step of the design process.

There are four common phases of development: Requirements, Design, Implementation, and Testing. Security is vital in each phase.

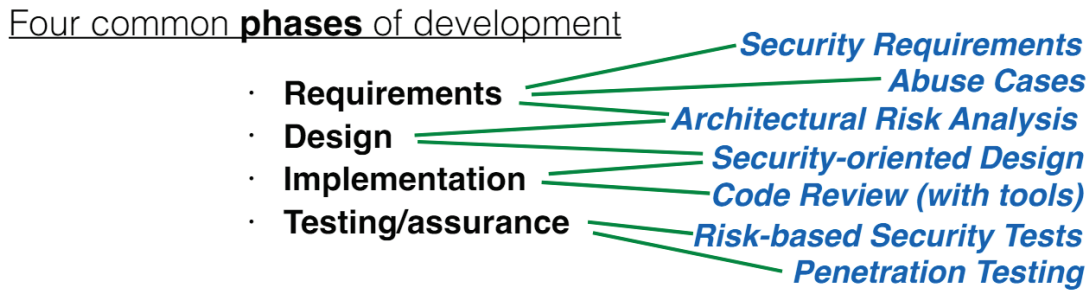


Figure 13: Diagram showing the role of Security in each step of the Design Process

The main thing to remember is that designing secure systems is a three step process: First, you model the anticipated threats, then define your security requirements, and lastly you apply good security design principles.

The threat model is arguably the most important portion of the secure design process. A threat model makes **explicit** the potential adversary's assumed powers. As a consequence, the threat model must be as close to reality as possible to minimize the risk of making mistakes in the risk analysis process. The threat model is also **critical** because lack of explicit knowledge about the attacker and their capabilities renders any analysis conducted on a software inaccurate.

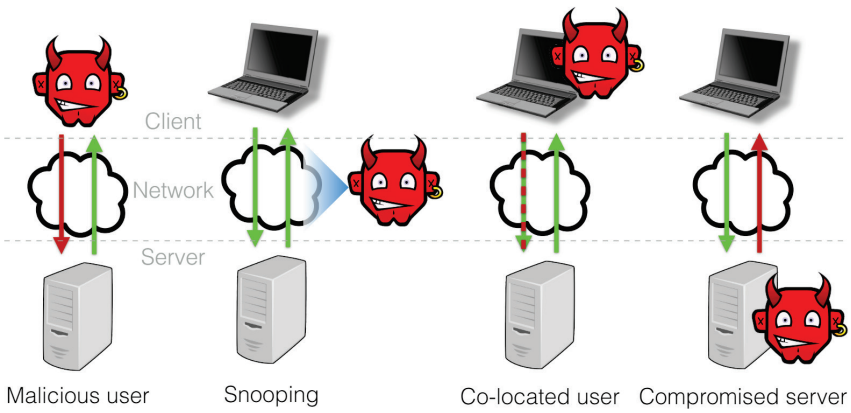


Figure 14: Graphic showing the possible network threat models

Recall that the threat model defines, in an explicit manner, the assumed capabilities of any potential attacker to the software. Threat-driven Design makes sure to incorporate the threat model

into design of the software by designing a different response based on the perceived threat model. If the threat model is that there is a malicious user, then there is no need to encrypt communications along the network since message traffic is implied to be safe. In the event that the threat model is of a snooping attacker, the best practice is to use encrypted Wi-Fi, encrypted networks, or encrypted applications. Lastly, if the assumed threat is that there is a co-located attacker (client-side or server-side), then it is assumed that the attacker can access local files and memory, meaning that unencrypted secrets should not be stored. Because the threat model is an **assumption** of a presumed attacker's capabilities, any assumptions made are potential holes that can be exploited by the attacker. The best way to find a good model is to compare with similar systems, understand past attacks, and challenge common assumptions when designing.

Now that we have a threat model, we can use it to define what we want to defend against. Software requirements are most often about what the software should do. In secure design, we want to have security requirements which are security policies, combined with the required mechanisms for enforcing them. Typical security policies use some combination of Confidentiality, Integrity, and Availability in addition to supporting mechanisms such as Authentication and Encryption. The three main supporting mechanisms are Authentication (How a system knows **who a user is**), Authorization (How a system knows **what a user is ALLOWED to do**), and Auditability (How a system can tell **what a user DID**). Typically, security requirements are illustrated by what are known as abuse cases. A use case is a case describing what a system **should** do, while an abuse case is a case describing what a system **should not** do. To construct abuse cases, the main practice is to construct use cases in which an adversary's exercise of power could violate a security requirement.

Secure Design Principles fall into three main categories: Prevention (eliminate software defects entirely), Mitigation (reduce the harm from exploitation of unknown defects), and Detection/Recovery (Identify and understand an attack, in addition to undoing any changes). There are 11 general rules of thumb that, when neglected, result in design flaws that can be exploited.

- Security is economics
- Principle of least privilege
- Use fail-safe defaults
- Use separation of responsibility
- Defend in depth
- Account for human factors
- Ensure complete mediation
- Kerkhoff's principle
- Accept that threat models change
- If you can't prevent, detect
- Design security from the ground up
- Prefer conservative designs
- Proactively study attacks

Figure 15: Graphic of Bulleted List of the 11 general rules of thumb for Secure Design

- Security is Economics => You cannot afford to secure against everything, so secure only against the things that will have the greatest "return on investment" for an attacker.
- Principle of Least Privilege => Give a program the access it legitimately needs in order to function and NOTHING MORE.
- Use Fail-Safe Defaults => Things will inevitably break. When they do, they should break safely.
- Use Separation of Responsibility => Split up privileges so that no one person or program has total power.
- Defend in Depth => Use multiple redundant protections that ensure security is breached if **all of them** fail.
- Ensure Complete Mediation => Make sure **every access to every object** is run through a reference monitor.
- Account for Human Factors => Users must buy into the security model
- Account for Human Factors => The security system must be usable (make it easier rather than harder)
- Kerkhoff's Principle => Do NOT rely on **security through obscurity**

These are vital to building secure systems and should NOT be violated as much as possible.

An example of Secure design done well is a software known as VSFTPD (**V**ery **S**ecure **FTPD**).

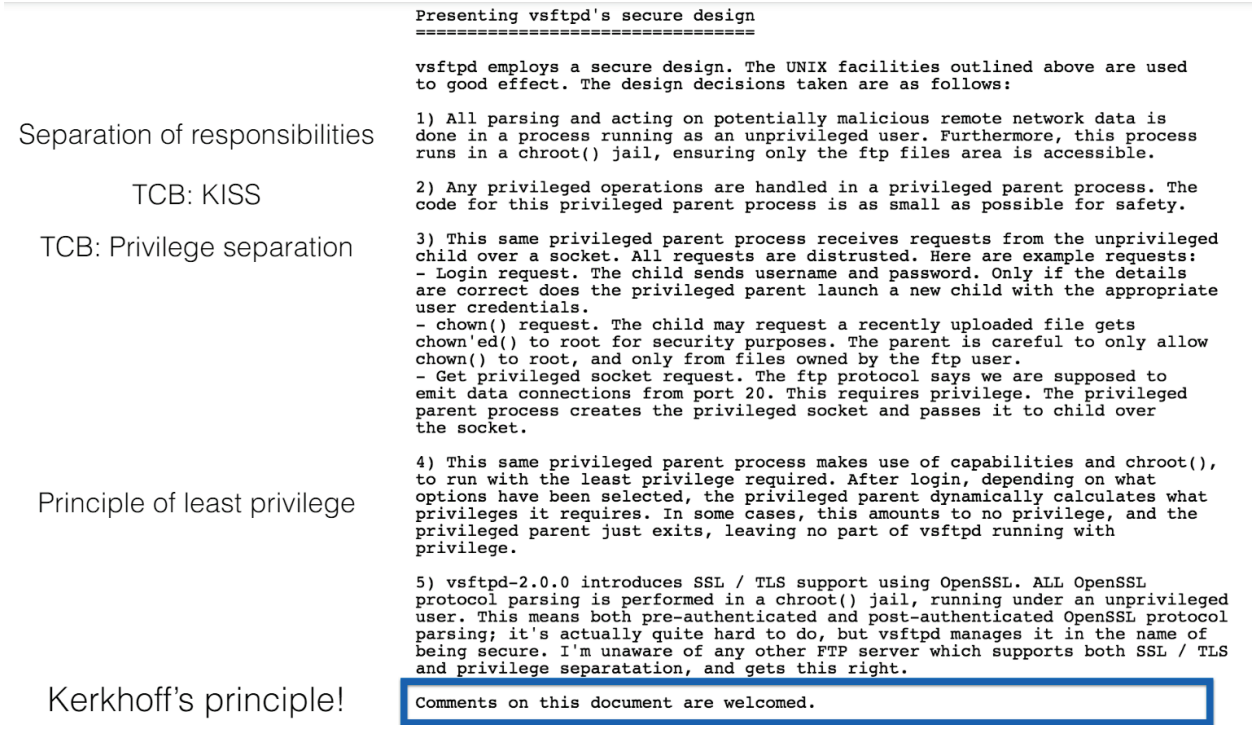


Figure 16: Snapshot showing VSFTPD Secure Design and the associated Secure Design Principles

AI/ML Security

All machine learning algorithms fall into four broad categories: Supervised Learning (Prediction and Classification), Unsupervised Learning (Clustering, Probability, Association, Dimensionality reduction), Semi-Supervised Learning, and Reinforcement Learning.

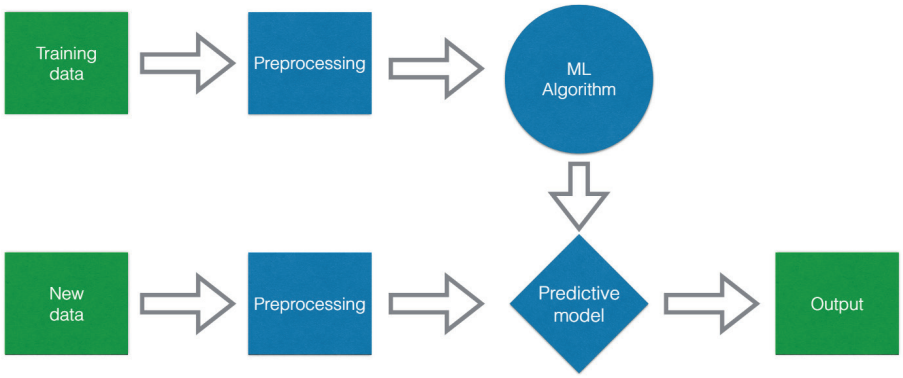


Figure 17: Simple diagram showing the overall ML workflow

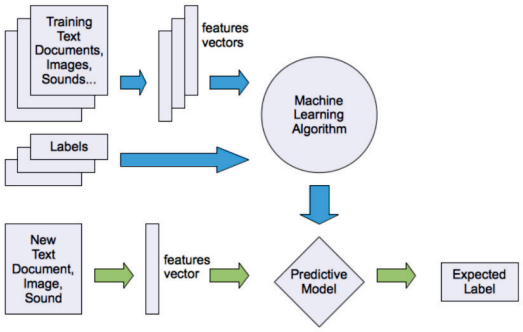


Figure 18: Simple Diagram showing the overall Supervised Learning workflow

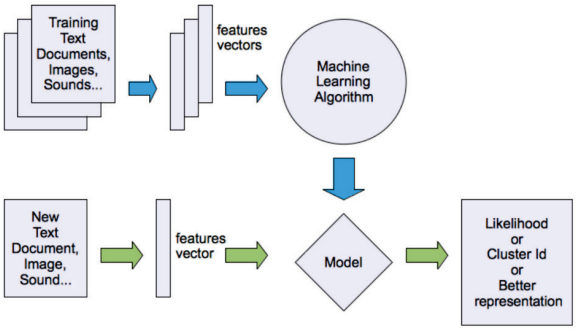


Figure 19: Simple Diagram showing the overall Unsupervised Learning workflow

Just as with any other system, Machine Learning models are vulnerable to attacks. Some common attacks on ML models include Evasion Attacks (perturbation), Poisoning Attacks, Model Inversion, and Backdoor attacks. In an Evasion Attack, an attacker attempts to cause a misclassification. The attack strategy depends on knowledge of the classifier such as the learning algorithm, feature space, and training data. Like Evasion Attacks, poisoning attacks also require advance knowledge of the model being used. However, unlike evasion attacks which attack the new data being used, poisoning attacks attack the training data in an effort to manipulate the model. The two most common poisoning methods are **injection of mislabeled samples** in the training data and **uniform alteration of worker behavior** by enforcing system policies. The third kind of attack is called a Model Inversion Attack, an attack in which **private** and **sensitive** inputs are extracted by using the outputs and the ML model. The second kind of attack involving the ML model itself is the model extraction attack, where the internal workings of the model are revealed by querying the model itself. The last and arguably most dangerous attack is a backdoor attack on a DNN. In this attack, hidden behavior is trained into a DNN, which runs its now modified training algorithm, on non-tampered data and vastly misclassifies those with the trigger present. However, these attacks are far too simplistic for real-world authors, who often add various layers of evasion functionality in order to bypass virus detectors provided by anti-virus software.

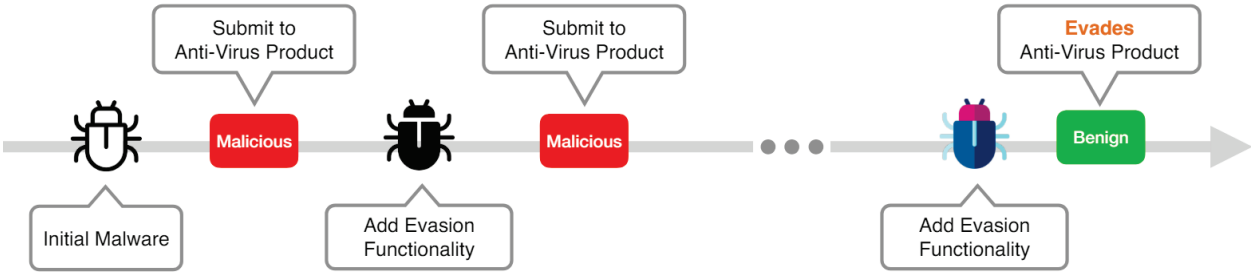


Figure 20: Timeline of making Malware register as Benign on an Anti-Virus Software (Norton, McAfee, etc.)

Symmetric Key Cryptography

Symmetric Key Cryptography, often referred to as secret key cryptography, is a fundamental concept in the field of information security. It plays a pivotal role in ensuring the confidentiality and integrity of data exchanged over insecure channels. In this exposition, we will delve into the principles, mechanisms, and practical applications of symmetric key cryptography.

At the heart of symmetric key cryptography lies the notion of a shared secret key between communicating parties. This key serves a dual purpose: encryption and decryption. The encryption process involves scrambling plaintext into ciphertext using the shared key, while decryption reverses this process, transforming ciphertext back into plaintext.

The strength of symmetric key cryptography hinges on the secrecy and randomness of the shared key. Therefore, the key must be kept confidential and changed periodically to mitigate the risk of compromise. Additionally, the key space, i.e., the total number of possible keys, should be sufficiently large to thwart brute-force attacks.

Symmetric key cryptography employs various algorithms and techniques to achieve secure communication. One such algorithm is the Advanced Encryption Standard (AES), which is widely adopted due to its robustness and efficiency. AES operates on fixed-size blocks of data and supports key lengths of 128, 192, or 256 bits.

Another notable mechanism is the Data Encryption Standard (DES), although its usage has waned due to its susceptibility to brute-force attacks. DES operates on 64-bit blocks of data and utilizes a 56-bit key. Despite its vulnerabilities, DES laid the groundwork for modern symmetric key algorithms.

Consider Alice and Bob, two parties wishing to communicate securely over an insecure channel. To establish a secure connection, they first agree upon a secret key using a secure channel or a trusted third party. Once the key is established, Alice encrypts her message using the shared key and sends the ciphertext to Bob. Upon receiving the ciphertext, Bob decrypts it using the same key, thereby retrieving the original message. Just as with any secure communication, recall that we are trying to maintain the CIA of Network Security (**C**onfidentiality, **I**ntegrity, and **A**uthenticity). Essentially, regardless of what encryption tactic we use in the creation of the symmetric key, we want to be able to keep others from reading the messages or data communicated (Confidentiality), we want to keep others from *imperceptibly* tampering with the messages or data communicated (Integrity), and we want to keep others from *undetectably* impersonating either of the communicating parties (Authenticity).

In the realm of symmetric key cryptography, several essential "black boxes" play instrumental roles in securing communication and data integrity. These black boxes encompass Block Ciphers, Message Authentication Codes (MAC), Hash Functions, and the Diffie-Hellman Key Exchange. Each serves a distinct purpose, contributing to the overall robustness and efficiency of cryptographic systems.

Block Ciphers are cryptographic algorithms that operate on fixed-size blocks of data, transforming plaintext into ciphertext using a shared secret key. They provide confidentiality and are integral to symmetric encryption schemes such as AES (Advanced Encryption Standard) and DES (Data Encryption Standard). **Message Authentication Codes (MAC)** ensure data integrity and authenticity by generating a tag, or MAC, which is appended to the message. This tag is computed using a secret key and the message itself, allowing recipients to verify the integrity and origin of the received data. HMAC (Hash-based Message Authentication Code) is a popular MAC construction based on cryptographic hash functions. **Hash Functions** are one-way functions that map arbitrary-sized input data to fixed-size output, known as hash values or digests. These functions are employed in various cryptographic applications, including data integrity verification, password hashing, and digital signatures. A robust hash function exhibits properties such as collision resistance and preimage resistance, ensuring the security of cryptographic protocols. **Diffie-Hellman Key Exchange** is a cryptographic protocol used to establish a shared secret key between two parties over an insecure channel. Unlike symmetric key exchange methods, such as pre-shared keys, Diffie-Hellman enables parties to negotiate a shared key without prior communication. This protocol relies on the discrete logarithm problem for its security and forms the basis for many secure communication protocols, including SSL/TLS. Together, these black boxes form the building blocks of symmetric key cryptography, providing the necessary tools to achieve confidentiality, integrity, and authenticity in digital communication. Understanding their principles and functionalities is essential for designing and implementing secure cryptographic systems in practice.

Symmetric key cryptography is a cornerstone of modern information security, facilitating secure communication and data protection. By leveraging shared secret keys and robust encryption algorithms, it enables parties to exchange sensitive information with confidence. However, its efficacy depends on proper key management practices and the judicious selection of encryption algorithms. As adversaries continue to evolve, so must our cryptographic techniques to ensure the continued security of digital communications.

Public Key Cryptography

However easy and quick symmetric-key cryptography may be it has three main shortcomings. Firstly, it requires pairwise key exchanges, which can run as high as $O(n^2)$ if the network being studied is an all-to-all network such as a chatroom or email service. Secondly, the use of symmetric-key cryptography requires that both users be online in order to preserve authenticity. An example is if one user uploads a document to a popular document hosting site and then goes offline *forever*, no other user (apart from the original uploader) can be certain that the document was really uploaded by the uploader. Lastly, using black-box methods such as Diffie-Hellman prevents eavesdropping, preserving confidentiality. However, these methods are not resilient to tampering meaning that integrity is not preserved.

Public Key encryption comprises three separate algorithms. Firstly, the key generation algorithm used is often a *randomized* algorithm with a **nondeterministic** output making it difficult to infer the Secret Key (key known by only one person) from the Public Key (the key shared with all users on a network). The thing that makes public key encryption slightly stronger than symmetric key encryption is that the public and secret keys are bound together in such a way that for every public key, there is **a single** secret key. Secondly, the message is encrypted using the **public key** of the intended recipient. This step takes a message of a certain length, encrypts it using the public key of the intended recipient in some way, and then returns a ciphertext that is **the same length** as the original message. The last step in public key encryption is the decryption itself. The decryption depends on the recipient's **secret key** which, in an ideal situation, should only be known by the intended recipient. This algorithm, unlike the previous two steps, **IS** deterministic, meaning that any ciphertext, if decrypted correctly, should return the original message. Furthermore, to ensure security when using public-key cryptography, the encryption method should appear almost random meaning that small changes in the plaintext should, theoretically, result in large changes in the ciphertext. All things considered, the encryption function used should approximate somewhat of a one-way trapdoor (a function with no backdoor that cannot be decrypted without the secret key used).

An alternative to both public-key cryptography (it can often be slow) and symmetric-key cryptography (it may not be very secure) is the use of what is known as Hybrid Encryption. Hybrid encryption follows almost the same process as public-key encryption, except that it adds an extra step in which a symmetric key is generated and used for the decryption as opposed to a secret key. In this way, it combines the best attributes of both methods by preserving the security and authenticity of the public-key encryption and combining it with the speed and randomness of the symmetric-key encryption.

Just as with all encryption, the goals of public-key encryption are not only to preserve the content and integrity of the message, but also to be able to define with certainty the sender of the message. In public key encryption, the authenticity of the messages sent on a network are determined by a set of digital signatures which use the SECRET KEY, meaning that only the sender can ever verify a message they sent.

Public Key Infrastructure

The main question to answer here is “How does a user know **for sure** who they are communicating with?” Let’s look at an example that will provide some insight into the matter. Let's say you’re on your favorite browser, Google Chrome, and that it’s your birthday. You want to see how much “Birthday Money” your relatives have sent you, so you log in to your bank, Bank of America. Now how do you know that you’re communicating with Bank of America itself and not a mockup? This is where a digital signature comes in. Bank of America has a Public Key that it sends to a Certificate Authority (let’s assume it’s VeriSign in this case). The Certificate Authority will then attempt to verify the authenticity of the public key and (let’s assume it’s a correct key) issue a certificate. Behind the scenes, your browser (Chrome) receives the key and certificate from Bank of America verifying that the website is legitimate. This is an instance of what is known as the Public Key Infrastructure or PKI. Now Bank of America is verified as legitimate by the certificate from VeriSign. However, can you trust VeriSign? This is the second step of the PKI in which the Certificate Authority (VeriSign in this case) is certified by another Certificate Authority (let's use Symantec as the second CA). Now how do you know whether Symantec is legitimate? This is the third and final step of the PKI, known as a root certificate, in which the certificate authority provides **self-verification** which is often irrefutable.

Figure 21: The PKI and Verification Process (Symantec is what is called the Root Certificate)

However, there are many root certificates and tracking them all can be difficult or even impossible. For this reason, all modern devices have a **single** root key store, which stores all root certificates. **THIS IS VITAL TO SECURITY, SO IT CANNOT CONTAIN ANY MALICIOUS CERTIFICATES.**

The possibility of malicious certificates raises an interesting question: “What happens if a certificate becomes invalid for any reason?” This is the beginning of a process known as certificate revocation in which the website asks the CA to revoke its certificate with an intent to reissue an updated certificate in the future. This is vital to security and in order to be as safe as possible, websites should request revocations as quickly as possible whilst browsers should check for revoked certificates constantly, obtaining revocations as soon as possible. This fact becomes more

prevalent when considering an attack like Heartbleed. When the "Heartbleed" issue was discovered, every vulnerable website should theoretically have patched, revoked, and reissued their certificates. In this way, Heartbleed sort of became an industry experiment on how quickly and thoroughly administrators react to issues. With regards to certificates, events were described as "Heartbleed induced" if the certificate was reissued on or after April 7th, had a validity of more than 60 days, and the domain reissued the certificate less than once every two months. What this testing showed was generally positive as only 7% of domains in the Alexa dataset were still vulnerable 3 weeks removed from the patching. With regards to revocations, the statistics were somewhat concerning with 13% of websites revoking their certificates and 27% reissuing them.

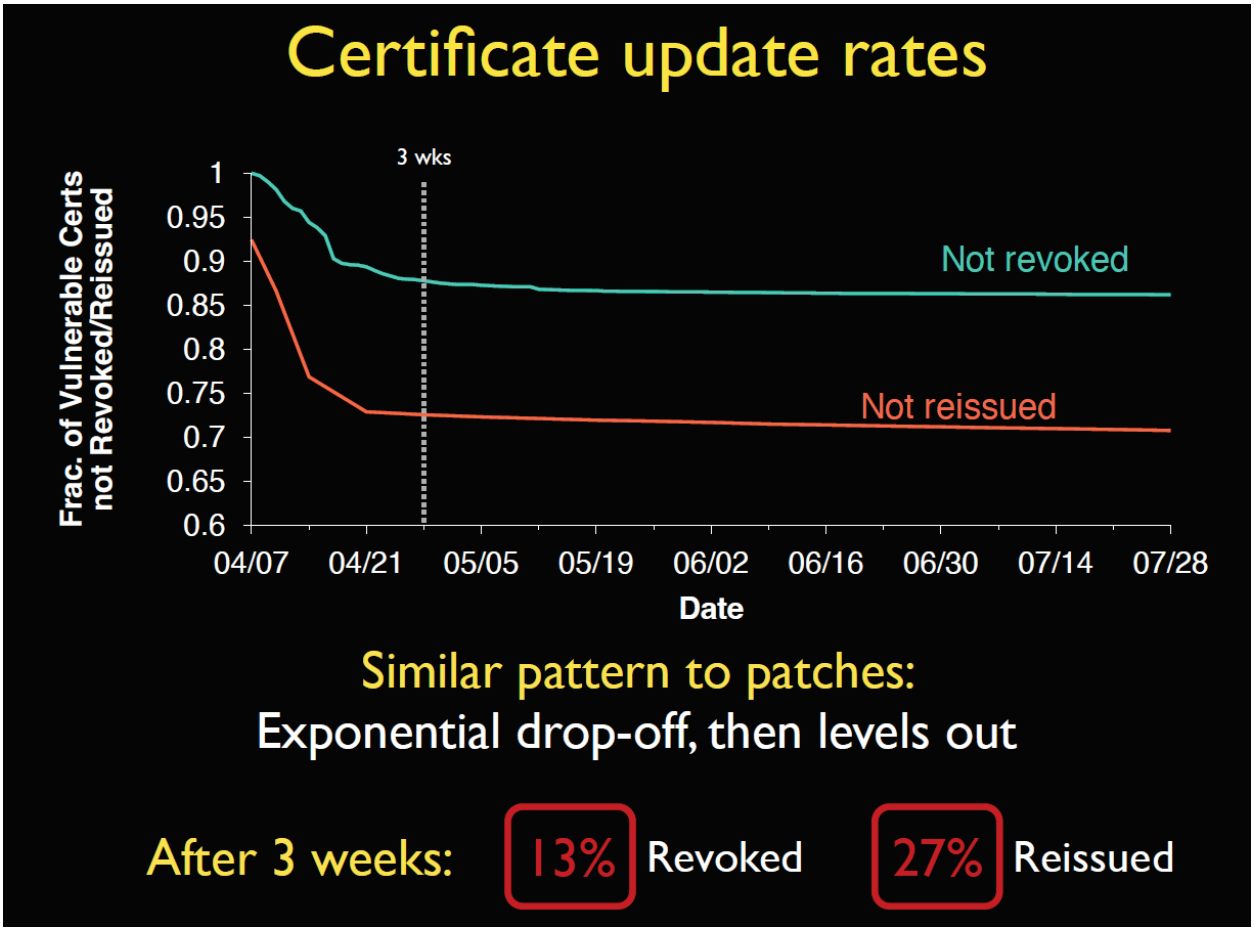


Figure 22: Graph showing the rates of Revocation and Reissue of certificates after identification of Heartbleed

Another somewhat concerning statistic is that of the certificates issued by various websites, 60% of retired certificates were never revoked. Currently, approximately 8% of vulnerable certificates still haven't expired, meaning that Heartbleed could be a continued problem for years to come.

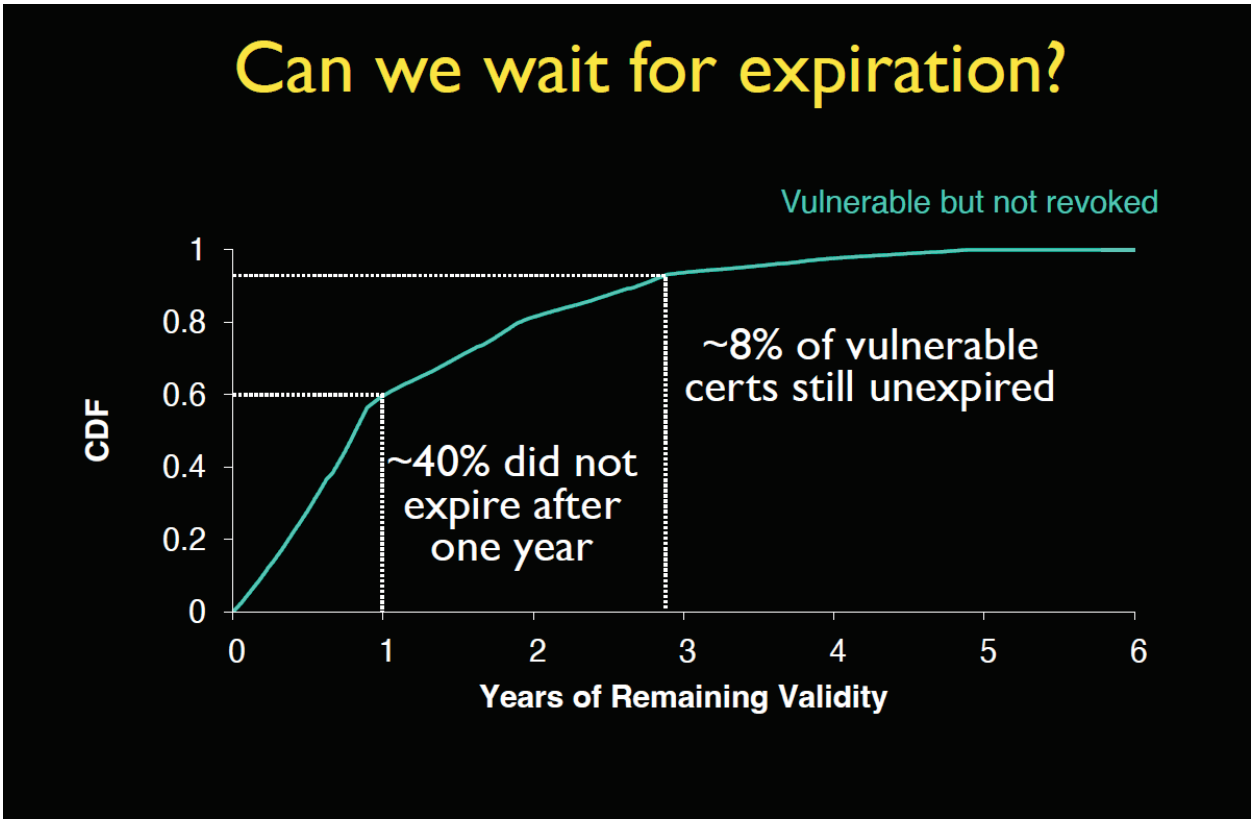


Figure 23: Graph showing the rate of Revocation for vulnerable certificates based on Expiration

The other concern with today's PKI (which we are all often visiting without even realizing) is the use of a CDN (**C**ontent **D**elivery **N**etwork), a third-party hosting service that has varying levels of involvement but is trusted for one single purpose: **delivering fast and accurate content.** The security concern here is that third-party CDNs know the private key of each of their customers, creating a security nightmare in the event that a CDN is hacked. The only way to verify the authenticity of a CDN is through what is known as a Subject Alternate Name (SAN) list. In spirit, this list is supposed to have *multiple names* for the ***same organization.*** However, the SAN list is often seen as multiple **unrelated** domains lumped together (since they use the same CDN).

How Security Fails in Practice

There is a well-known saying pertaining to security. It goes something like this: “A chain is only as strong as its weakest link.” Nowhere is this idea encapsulated more than in the failures of security in different real-world scenarios. For example, there are currently 15,134 apps on the Google Play Store that use some form of cryptography. Of these 15000+ apps, almost 11,800 were analyzed. In that analysis, it was revealed that nearly 5700 (about half) used a mode of encryption known as ECB. Now, we’ve all seen the famous image of Tux the penguin (the lovable Linux mascot) encrypted using ECB. If you look at it closely, you can see that it isn’t very encrypted at all. You can still tell that it is an image of the beloved Tux. This isn’t so much because ECB is bad (it is), but rather because ECB used a strong cryptographic technique (block ciphers) in a poor way.

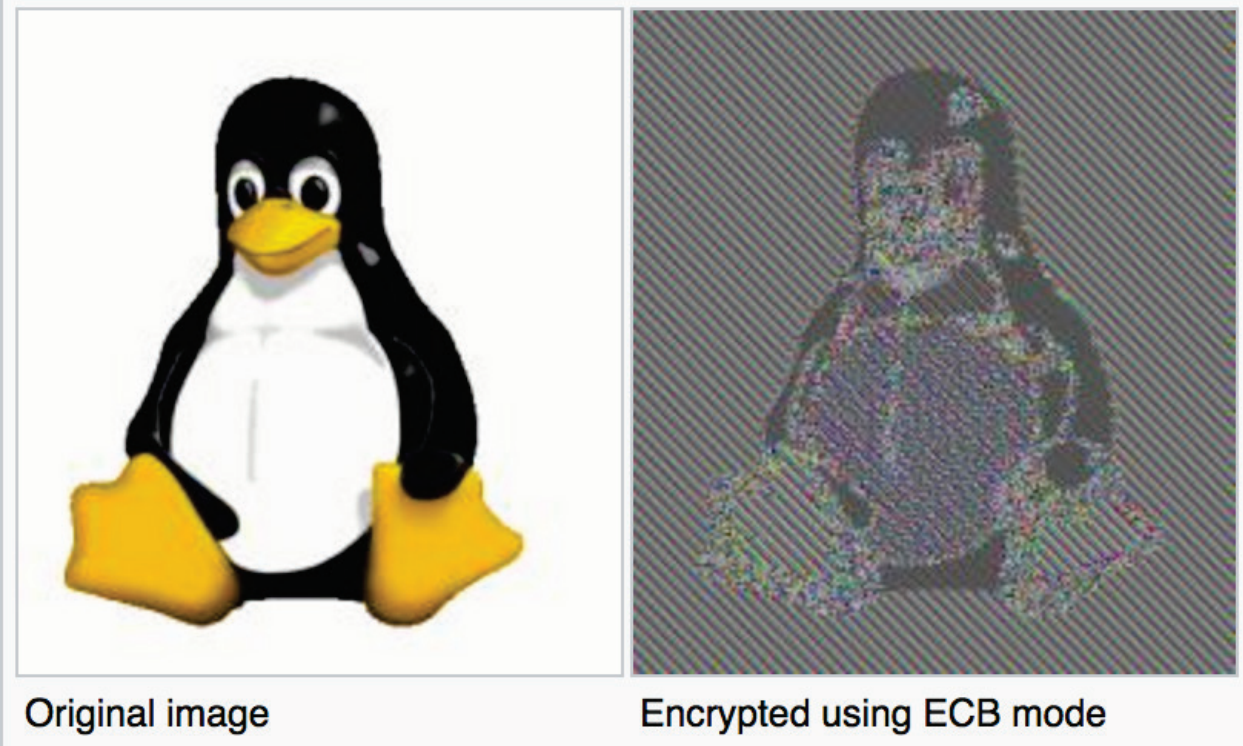


Figure 24: Tux the Linux Penguin Encrypted using ECB

Another way in which poor encryption can create a security headache is the use of a CDN. Often times, when using a CDN multiple companies using that CDN will all share a key. This makes CDNs a prime target for encryption-based attacks since they are all based on one key, meaning attacking a CDN can attack multiple companies, creating mass havoc.

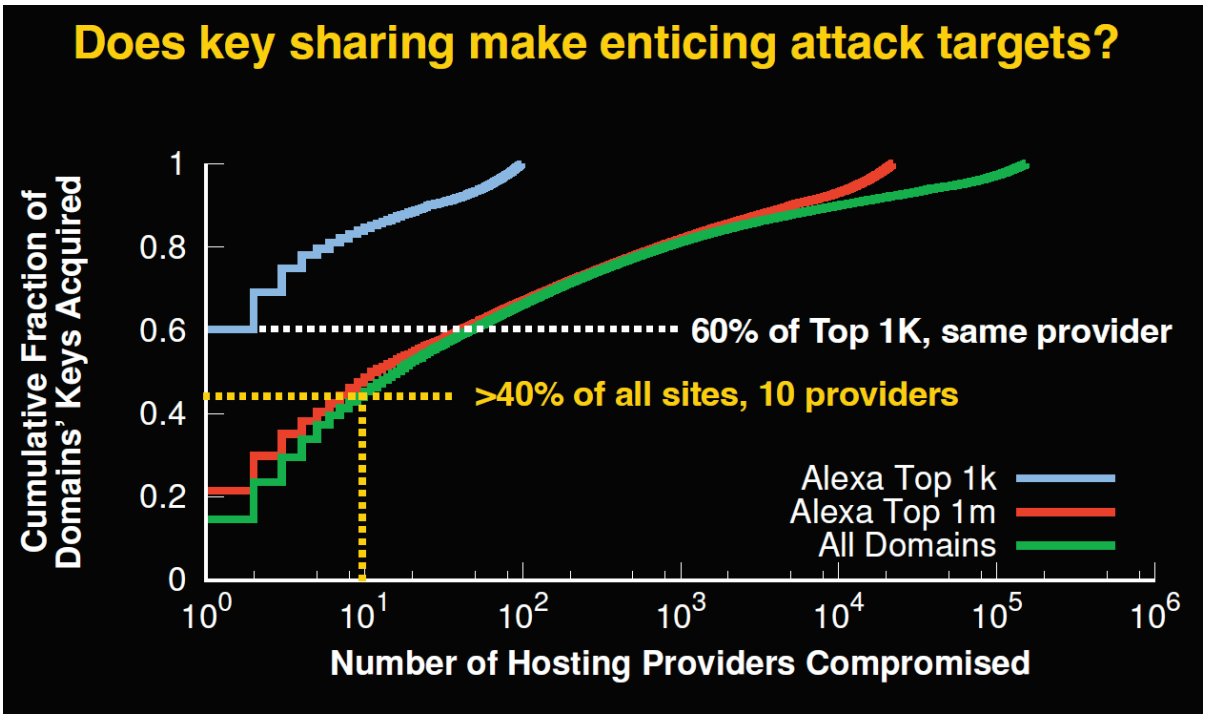


Figure 25: Graph showing how attacking a CDN can cause havoc on the internet

Anonymity

Let's imagine a scenario. You're James Bond, the famous 007. However, you're lost deep undercover on a mission, and you need help from headquarters. Because you're undercover, you don't want your identity to be revealed. Also at HQ, you want your request to be classified, so nobody knows what trouble you are in. This is the main purpose of anonymity in security. The first situation (you don't want to reveal your identity) is called sender anonymity whereby there could be a large number of senders and any possible attacker cannot reliably infer who the sender was. The second scenario (you don't mind revealing your identity in lieu of your troubles being private to your team) is called receiver anonymity whereby the sender is known but the receiver is unknown.

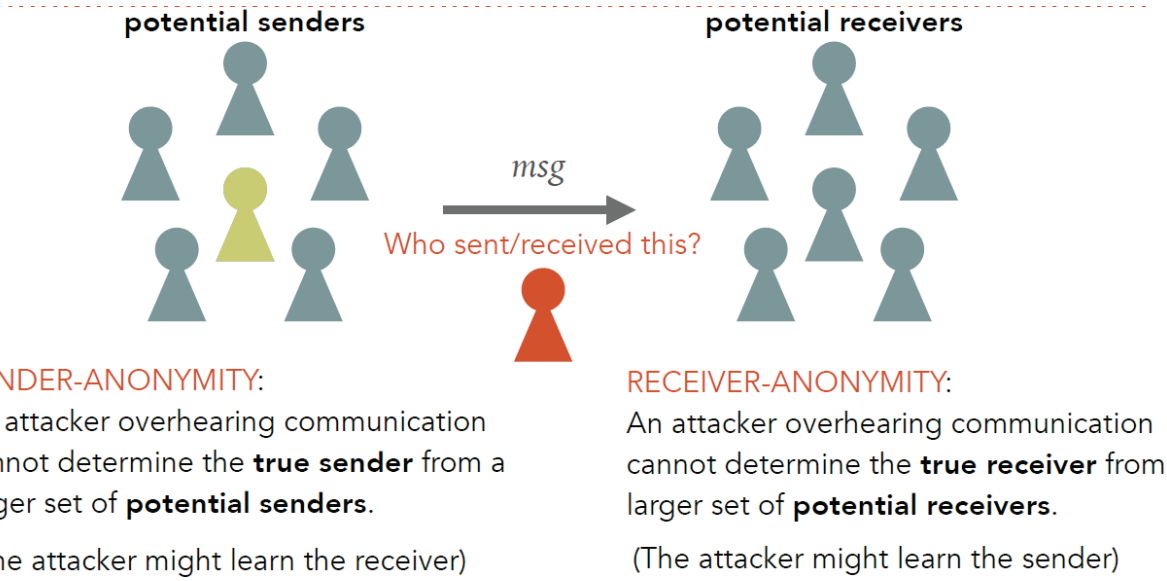


Figure 26: A More Thorough Explanation of Sender and Receiver Anonymity

As a last consideration of security, let's consider what are known as mix-nets (short for **Mix**er **Net**work**s**) In such a situation, let's assume that an attacker can eavesdrop on every possible link in the mix-net.

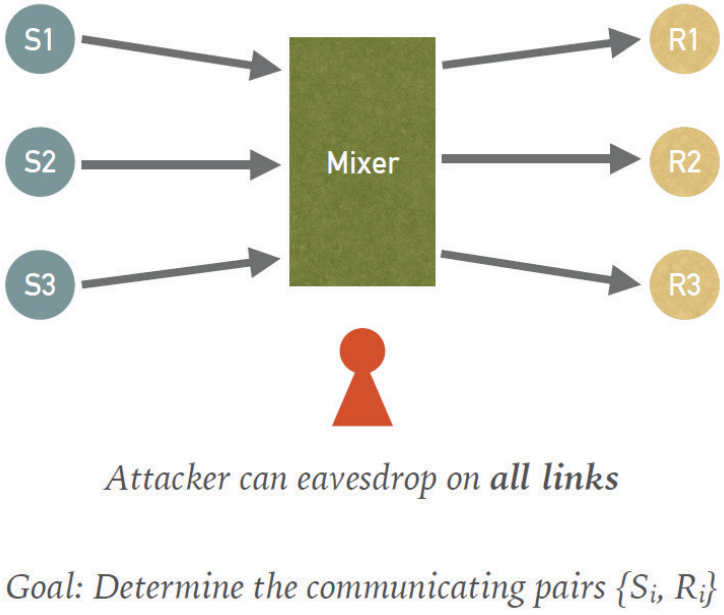


Figure 27: Simple Rendition of a Mix-Net (3 senders and 3 receivers)

In this case, sending all messages unencrypted would be catastrophic since there would be **absolutely no security**.

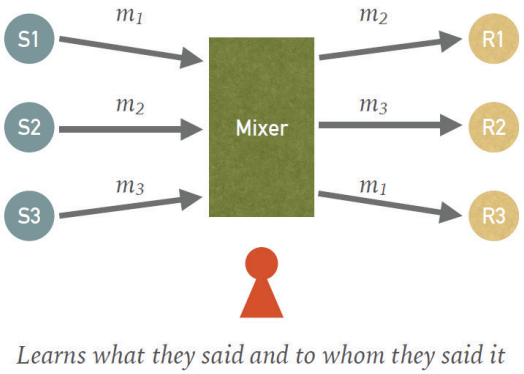


Figure 28: Diagram of the catastrophe of sending unencrypted messages on a compromised mix-net

Encryption may not be the best practice, but it can help if used correctly.

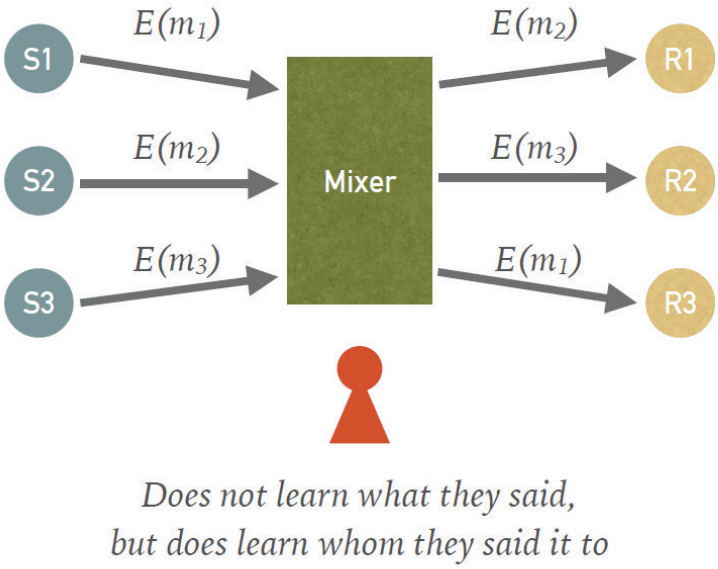


Figure 29: An example of encryption used POORLY in a mix-net

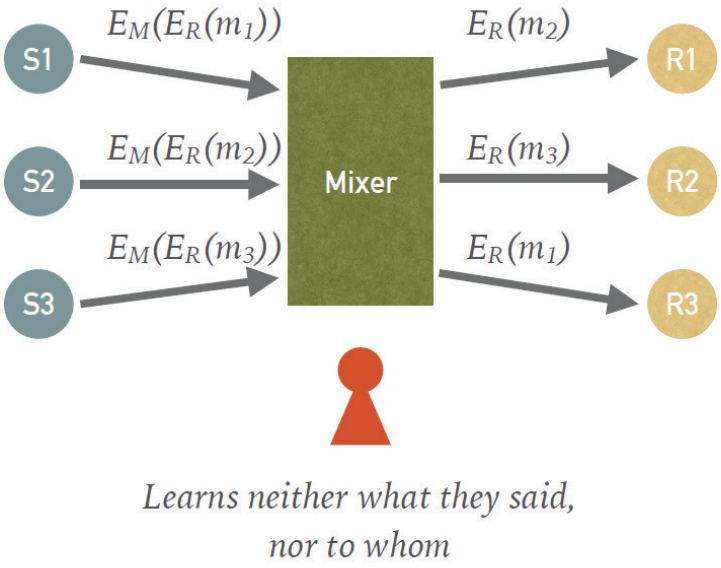
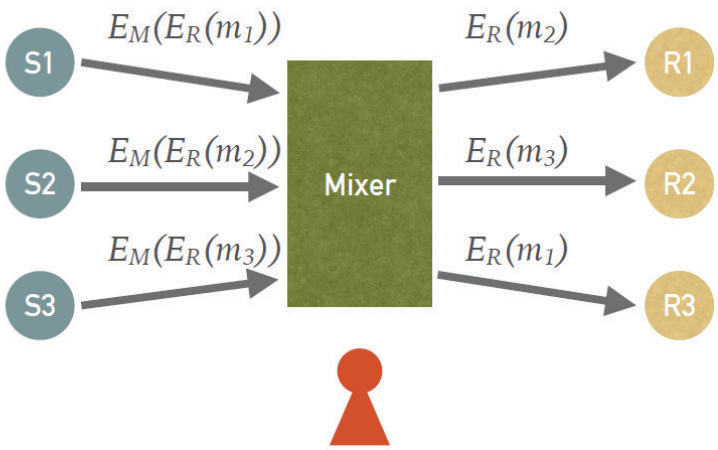


Figure 30: An example of encryption used WELL in a mix-net



What if the messages arrive at different times?

***Mix** the order of delivery*

Figure 31: An example of encryption used PERFECTLY in a mix-net (Delayed Delivery for messages arriving at different times)

Networking Basics

The internet is the heart of everything we do each and every day. It is a fascinating thing and understanding how it works is at the heart of network security. The internet runs based on what are known as protocols which are, simply put, agreements on how to communicate. These protocols are so vital, in fact, that they have been publicly standardized via Request for Comment (RFC) files. Essentially, coding a product based on the protocol of choice enables that product to connect with others on the internet. In addition, the internet works because the network is, as security engineers like to put it, dumb. In principle, the routers (internal nodes) have no knowledge of any connections that pass through them on the way to the final destination. Routers typically only perform destination-based routing and forwarding, whereby they send packets to the next hop along the circuit that is best suited to help the packet get to its destination. Because of this principle of routing, a good mental model for the internet is the Postal System. The internet is also robust because it is partitioned into layers, with each layer **providing** to the layer **immediately above** it and **receiving** from the layer **immediately below** it.

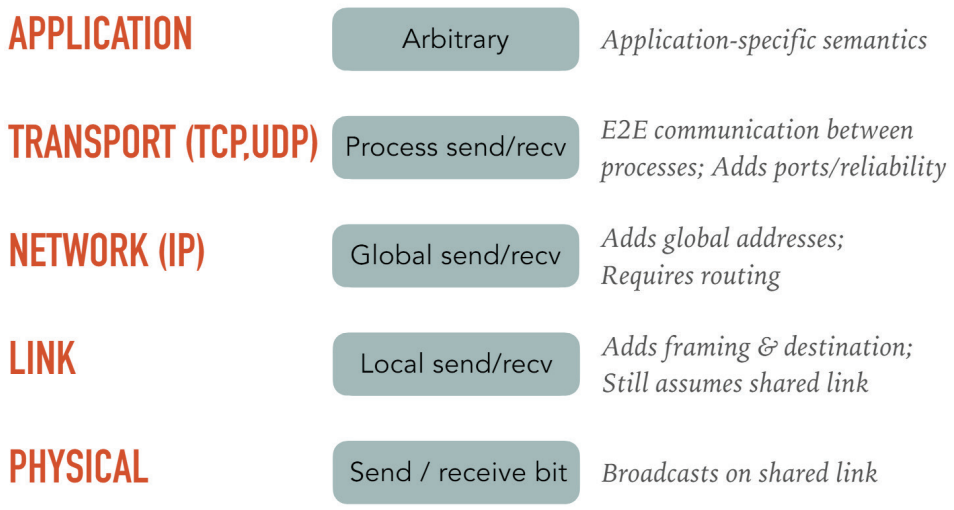


Figure 32: A rough enumeration of the layers of the internet and their associated functions

IP (In)security and VPNs

When communicating over any internet protocol, the things sent over the network are called packets. Each packet has a defined structure with a 20 byte header followed by some form of payload. The payload may itself be another packet, or even more header information however it is not usually known what the payload is.

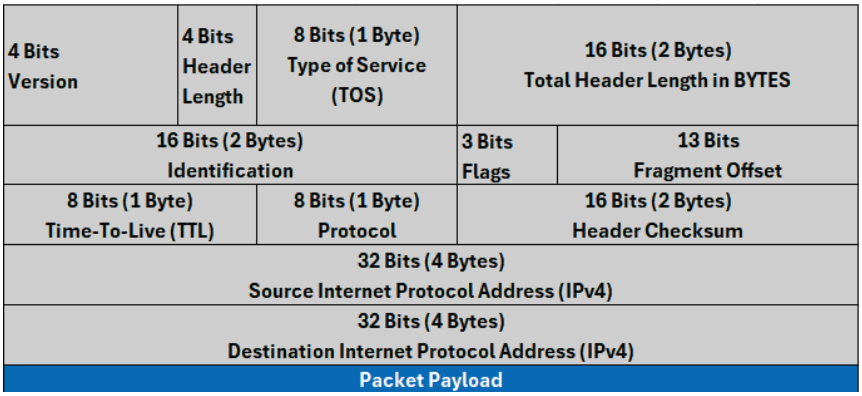


Figure 33: A Graphical representation of the contents of the header of an IPv4 Packet

Each of the fields in an IP packet has a different purpose.

- Version Number (4 Bits)=> Indicates the version (IPv4 or IPv6) and identifies, in a nutshell, the fields to follow
- Header Length (4 Bits)=> Indicates the number of 32 bit (4 byte) words that make up the header
    - Usually 5
- Type of Service (8 Bits [1 Byte])=> allows for different treatment of packets based on the service contained therein (audio vs video vs file transfer)
- Destination IP Address (32 Bits [4 Bytes]) => Where the packet is going
    - Allows routing nodes to make forwarding decisions
    - **Unique** identifier for destination
- Source IP Address (32 Bits [4 Bytes]) => Who sent the packet
    - Recipient can accept or decline packet
    - Recipient can reply to source
    - **Unique** identifier for source

Just as with any other network-based service, there can be attacks on IP. One kind of attack is called a Source-Spoof attack. Since there is nothing in IP enforcing the ownership of a source IP address, the source IP address can be spoofed (changed) to be someone else's, leading to their network receiving traffic they never requested. Another possible attack on IP is known as an eavesdropping attack. In an eavesdropping attack, packet traffic can be seen or even altered because IP does not provide any protection of the packet's header or payload. Due to the ease of attacking networks using IP as the means of attack, defenses have sprung up to detect spoofed packets. One such defense is called Egress Filtering. Recall that in a network, the point at which traffic enters the network is called the Ingress Point and that the point at which traffic leaves a

network is called the Egress Point. In egress filtering, IP packets are dropped if the source IP address of a packet **leaving** the network (egress) is not a match for any of the IP addresses on that branch of the network. Even though this is a safe and secure solution, Egress filtering is not widely deployed in most networks.

Recall that IP is also vulnerable to eavesdropping and tampering due to a lack of security. As a countermeasure, we can deploy what is known as Secure IP on top of the already used IP. This is the principle behind a VPN (**V**irtual **P**rivate **N**etwork), which does this using the predominant method which is IPSec (**I**nternet **P**rotocol **Sec**urity). There are two modes of IPSec. IPSec can either function in transport mode in which the payload is encrypted but the headers are not, or IPSec can operate in tunnel mode in which both the payload and headers are encrypted. The way in which IPSec does this is to encrypt the IP Packet being sent and use the encrypted packet as the payload of a second IP packet. The VPN server receives the encrypted packet, decrypts it, and sends the payload as if it had been sent by the client without use of a VPN.

TCP Attacks and DoS

TCP promises that packets will be reliably transmitted in that they will all be transmitted in order, with minimum drops and latency. It does these through a series of acknowledgements from the destination server. These acknowledgements can be shown in what is known as a "waterfall diagram" which indicates the communication between the source and the destination over time.

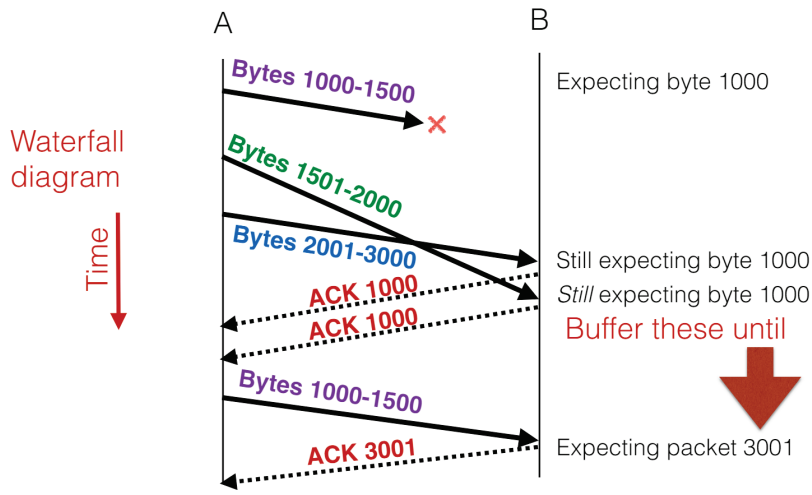


Figure 34: Waterfall Diagram showing communications between the Source (A) and Destination (B) regarding a message of arbitrary size split into packets of either 500 or 1000 Bytes

This is how TCP handles its promise to reliably transfer packets over the internet. TCP must also make sure that traffic being sent over the network does not consume too much capacity on the network itself. It does this by dynamically adapting the speed with which the source responds to acknowledgement packets (ACK) from the destination. Just as IP packets have a header, so do TCP packets. The TCP header is a bit different than the IP one as the first 32 Bit (4 Byte) word contains the Source and Destination port numbers as opposed to version information.

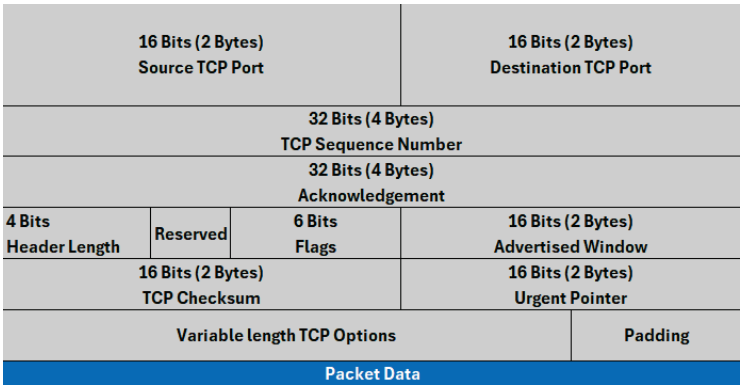


Figure 35: Basic Illustration of the contents of a typical TCP Packet

More often than not, this TCP header is usually placed under an IP Header and the two are transmitted together over the network.

The sequence number in a TCP packet is unique to the packet being transmitted but is shared between the sender and receiver. It is the sequence number of the first byte in the packet's data section. The next sequence number is the previous sequence number incremented by the data size of the previous packet's data. The acknowledgement in the header is the sequence number expected to be communicated by the opposite end-host (source => destination, destination=>source). In addition, TCP packets contain flags for maintaining the network and preventing an overload of network traffic. There are four main flags that are used: SYN (short for **SYN**chronize the network connection), ACK (short for **ACK**nowledged receipt of package, waiting for next packet), FIN (short for **FIN**ished communication, no further packets to send) and RST (short for **R**e**S**e**T** the connection because an error has occurred). Before beginning any communication over a TCP network, the intended source and destination must perform a three-way handshake which involves two **SYN** packets and two **ACK** packets.

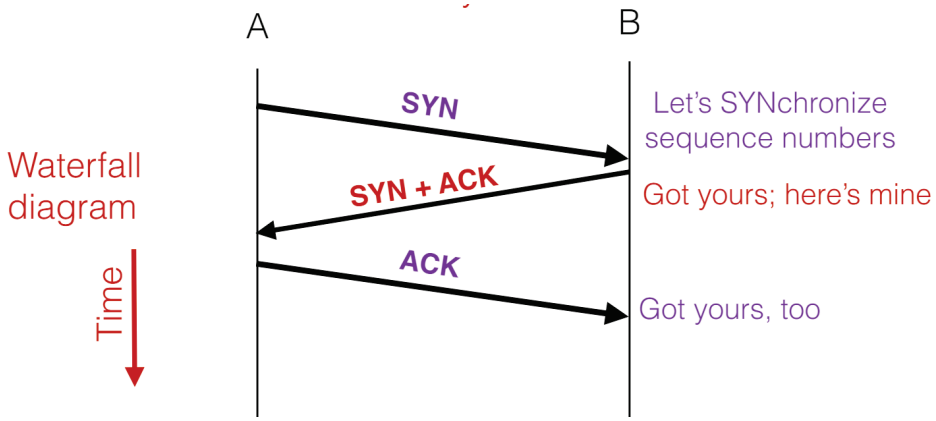


Figure 36: A waterfall diagram showing the process of initializing a TCP connection by way of a three-way handshake

Just as with any network or protocol, TCP has flaws that make it vulnerable to attacks. The first kind of attack is known as SYN flooding. SYN flooding is an attack in which a malicious source floods a victim (destination) with as many SYN packets as possible knowing that the machine will allocate state for each SYN it receives. In essence, the purpose is to exhaust memory on the victim server so as to prevent any other connections from being generated. In that way, SYN flooding is one type of DoS (**D**enial **o**f **S**ervice) attack.

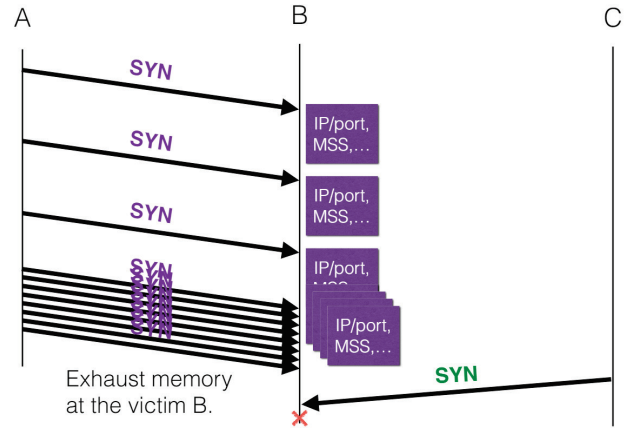


Figure 37: Triple waterfall diagram showing the workings of a SYN Flooding attack, a type of DoS attack

As is the accepted practice for any attack, there must be a defense. The defense for a SYN flooding attack is to use what is known as a SYN cookie. For each SYN request that a host receives, it creates a cookie based on the state and data provided in the request and sends it back in its SYN-ACK packet in the response to the source. The state on the destination host will only be allocated for a given source if the cookie sent by the source matches the one the destination had created earlier in the process.

Another type of attack that can be launched on TCP is what is known as an Injection Attack. The most common Injection attack launched against TCP is called the Mitnick Attack. In a Mitnick Attack, the trusted server is SYN flooded first. Its IP address is then spoofed in a SYN packet sent to the source. The source, believing the IP address is legitimate, sends a SYN packet to the IP address listed, which the attacker can now see. The response is then spoofed as the proper ACK is generated using the recovered sequence number. The last step is to clean the trusted server by redirecting all traffic to the spoofed IP address while simultaneously sending a RST (Reset) packet to the original server. The only known defense against such an attack is to make the initial sequence numbers have a degree of randomness to them.

The final type of attack that can be launched on TCP is what is known as an Optimistic Acknowledgement (OPT-ACK) attack. Recall that for data to be sent, each prior packet must be acknowledged as received before the subsequent packet can be sent. This kind of attack can be thought of as a form of DOS attack, but it is a DOS attack on the source as opposed to the destination. How it works is that due to the predictability of packet structure and sequence numbers, if the attacker can work out what the sequence number will be from the packet sent by the source, it can send the source an ACK (Acknowledge) packet before the source can send it any data. In this way, the source believes that it has a fast and legitimate connection to a destination until the source begins to lose packets due to the high speed of network traffic along the path from the source to the destination.

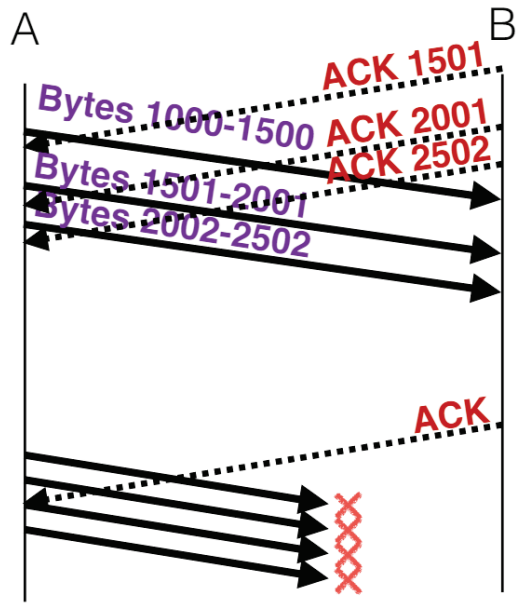


Figure 38: Waterfall Diagram showing OPT-ACK Attack initiated by attacker (B) on Source (A)

This attack is made even more dangerous by the small size of ACK packets, which can lead to a huge amplification factor due to the use of cumulative ACK packets in the attack. Assuming defaults for most systems (max window of 65536, MSS of 536), we get that the default amplification factor is a whopping 1336 times. In addition, the use of window scaling increases the amplification factor to as high as 22 Million times. Furthermore, a MSS of the minimum value of 88 further amplifies the attack, raising the amplification factor to 32 Million times.

The main problem regarding OPT-ACK attacks is that there is no concrete defense against them other than incremental development.

DNS and DNSSEC

Recall that the internet functions on various protocols and IP (Internet Protocol) is one of them. IP addresses allow for global connectivity, but they are a bit useless since users don't normally pick their own IP addresses nor are they expected to remember others. The DHCP (**D**ynamic **H**ost **C**onfiguration **P**rotocol) is used to set IP addresses for each and every device that is connected to the internet, and these IP addresses are mapped to names by the DNS (**D**omain **N**ame **S**ystem).

A new host on the network will likely not have an IP address and would also not know where to ask for one. The solution for this is to contact a DHCP server to try to find one on the local subnet. This initial request to the DHCP server is called a DHCP "discover", which the server responds to with an "offer" containing an IP address, DNS server, and gateway router. Should the device feel the offer is appropriate, it will send a request to the DHCP server for the IP Address offered in its response.

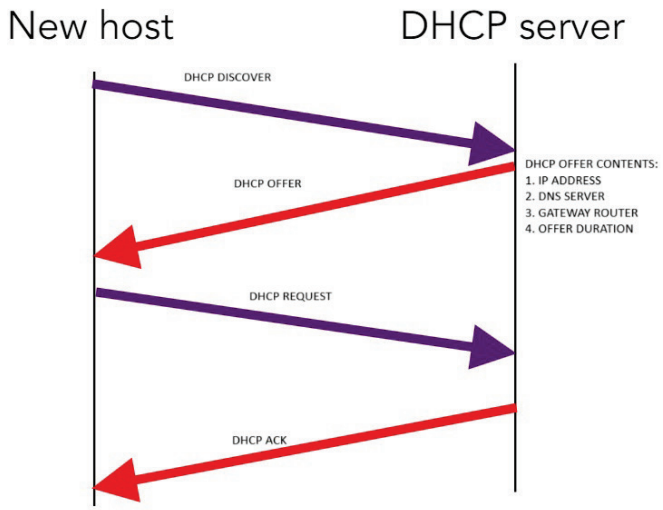


Figure 39: Waterfall Diagram showing the process by which a new device is given an IP address and a DHCP connection

Now, just like any other internet protocol, DHCP is vulnerable to various kinds of attacks. Since requests are issued as broadcasts, any attackers on a network can "hear" the request made by a new host. The attackers then race the actual DHCP server to replace the DNS for the requested website with one of the attacker's choice or even to modify the gateway used to send traffic so that all exiting traffic can be read at the egress point.

With DNS, the process undertaken by a DHCP is called Name Resolution, whereby the DHCP maps the IP address given with a memorable name. Names such as Google.com and ELMS.umd.edu are memorable but aren't routable unlike IP addresses such as 74.125.228.65 (Google's IP address). We can see what IP address a service uses by launching a **ping** to the associated server. If we wanted a DHCP "certificate" for a website, we can launch a **dig** on the website for which the certificate is desired.

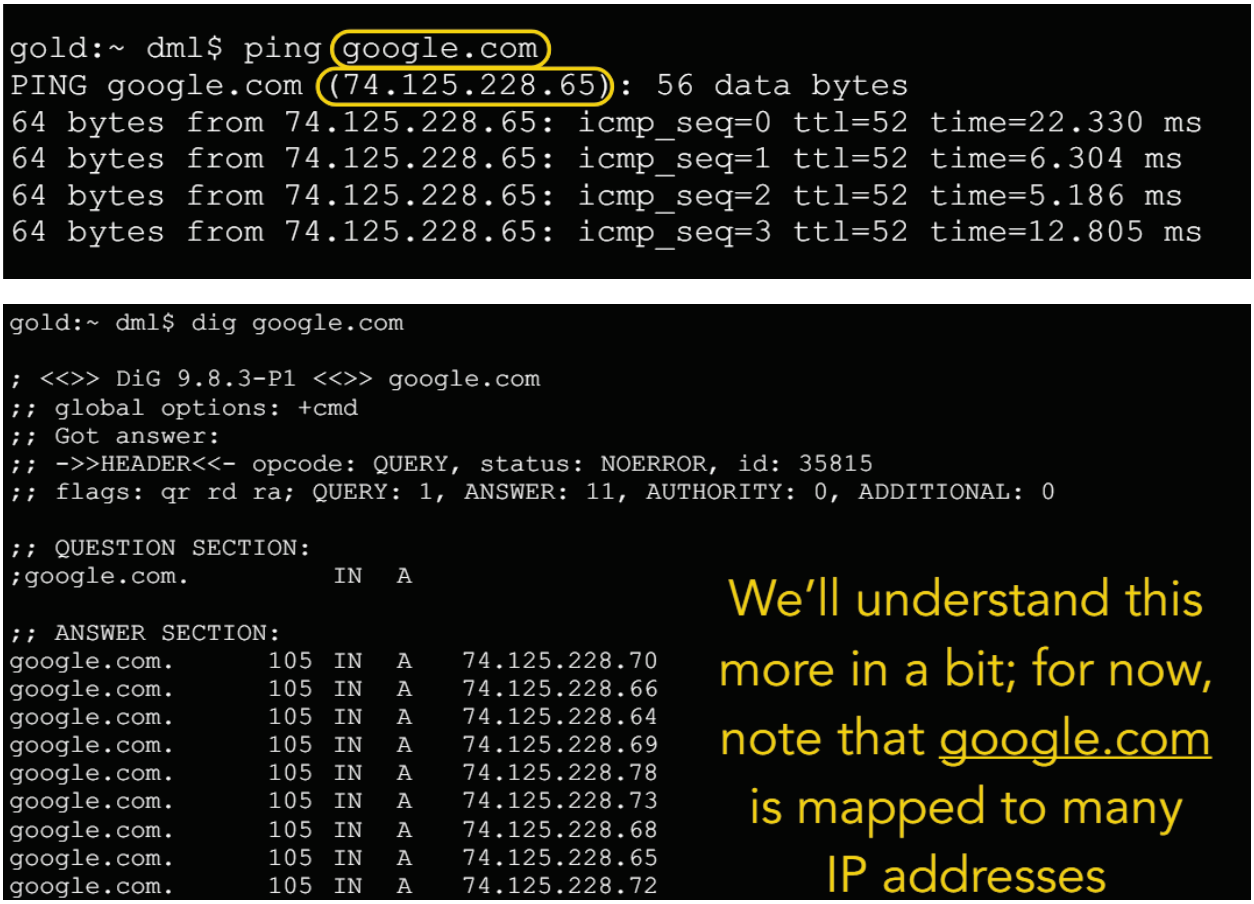


Figure 40: (a) Screenshot of Terminal showing ping to Google (b) screenshot of Terminal showing "dig" certificate of Google

There are a few terms to know with regards to DNS. Every DNS Namespace is divided into zones, which is done for administrative reasons. Essentially, a DNS zone is nothing more than a collection of Hosts/IPs that are lumped together purely by chance. In addition, subdomains do **not** need to be in the same zone, allowing for delegation of responsibility by a domain to its subdomain.

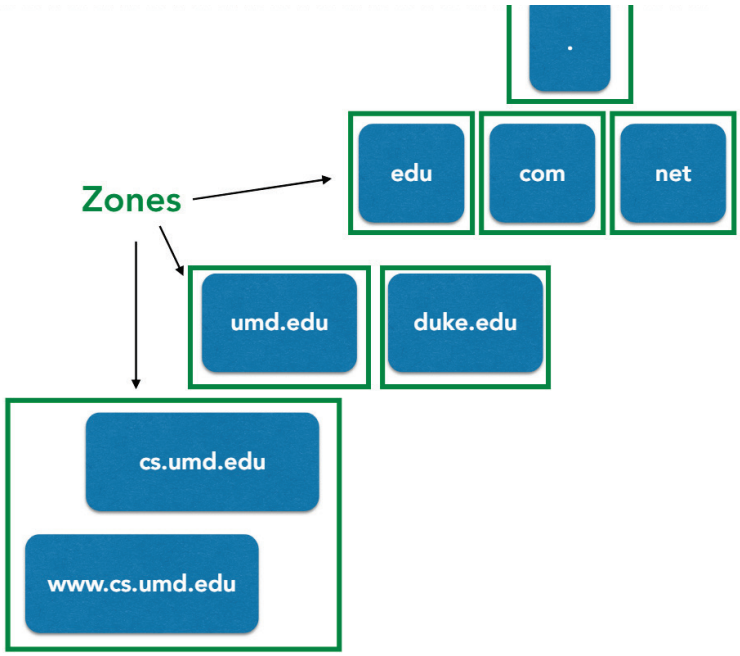


Figure 41: A Diagram showing a simple Namespace Hierarchy leading to www.cs.umd.edu

Namespace hierarchies give rise to what are known as nameservers. Nameservers are portions of code that answer any and all queries about a requested IP address, such as the owner and how to connect to it. In order to reduce latency and network load, each DNS zone must run at least two nameservers including an authoritative nameserver which is a file containing a complete and unabridged list of the addresses and hostnames in that zone of the network. While the nameserver answers DNS queries, those queries can come from one of many DNS resolvers. Every operating system has an inbuilt resolver, but that resolver is pretty weak. However, through the use of a recursive nameserver it can display many queries much faster than normal. Recursive nameservers are those that issue queries on behalf of a client resolver until a response (authoritative answer) is received. There is almost always a **local** recursive nameserver, but apart from those select few servers, nameservers do not normally support recursive querying. The requests sent and received on DNS are called Records. There are various record types, but the most common ones are the **A**ddress record, the **M**ail e**x**change record, the **S**tart **o**f **A**uthority record, and the **N**ame**S**erver record. Nameservers within any zone of a DNS network must be able to give Authoritative answers for hostnames within the zone and must be able to provide pointers to Nameservers who host zones situated in its subdomains.

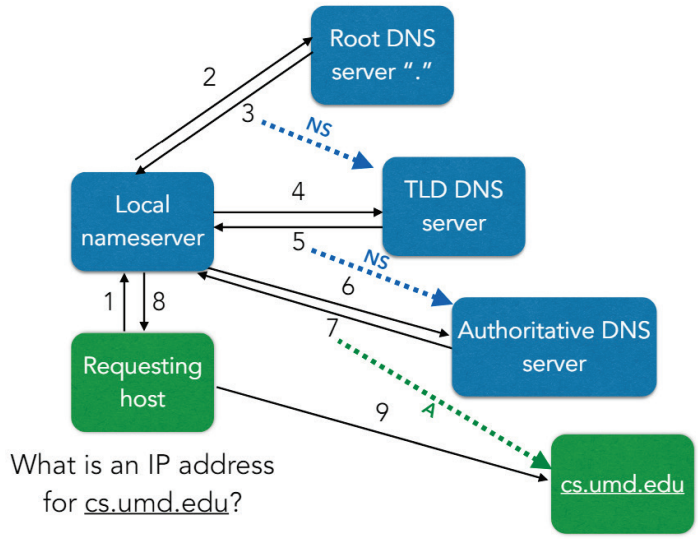


Figure 42: A diagram showing, at a high level, the process by which a host connects to cs.umd.edu over DNS

The address given in the A record in step 7 is known because the local DNS server learned it via DHCP, the parent server (umd.edu) knows the IP address of its children (which cs.umd.edu is) and the root nameserver is hardcoded into the local DNS server. Just as an interesting fact, UMD runs D-Root (the servers are named from A-Root to M-Root).

But just as with any other network-based service, DNS can be attacked. DNS often relies on cache to make sure that results are easily retrievable when needed, but this opens DNS up to what is known as a "cache poisoning" attack in which an attacker fills a victim's cache with false information.