

MIT 6.858 - Computer Systems Security

MIT 6.858 - Computer Systems Security

Taught by Prof. Nikolai Zeldovich at MIT in Spring, 2020.

6.858 Computer Systems Security is a class about the design and implementation of secure computer systems. Lectures cover threat models, attacks that compromise security, and techniques for achieving security, based on recent research papers. Topics include operating system (OS) security, capabilities, information flow control, language security, network protocols, hardware security, and security in web applications.

[6.858 on OCW](#)

[2020 course website](#)

[Lecture Videos \(YouTube\)](#)

Lectures

1. [Lecture 1: Introduction](#)
2. [Lecture 2: Security Architecture](#)
3. [Lecture 3: User Authentication](#)
4. [Lecture 4: Buffer Overflows](#)
5. [Lecture 5: Privilege Separation](#)

Lecture 1 - Introduction

This course is about building secure computer systems. Security is a property or behaviour of a system that is achieved despite attacks from adversaries. This property could be data confidentiality, data integrity, access control, etc.

There are three main aspects to building secure systems -

1. Policy - The plans and/or rules you will implement to make sure that your system is secure. For example, require users to enable 2FA.
2. Threat model - Assumptions about what the attacker can and cannot do. For example, the attacker can try to guess passwords, but cannot try to physically break into the server room (this assumption is fine in most cases).
3. Mechanism - The specific software and hardware techniques you will use to achieve security.

Building secure systems is hard, because security is a *negative goal*. For example, if our goal is to make sure that only the author of a file can access that file, implementing access control is a positive goal - only one code path is needed to check if the accessing user is the owner, and if yes, access is granted. However, to make sure that *no one else* accesses the file, we may need many checks, techniques, etc. to make sure that no one else is granted access to that file. We must make sure that there is no way in our system for a user that is not the owner to pretend to be the owner.

This is a very hard task, and there is no guarantee that we will get it right the very first time. There is also no guarantee that we have thought of and considered all the possible attack vectors for a system. Therefore, we must constantly be vigilant and iterate. We must be vigilant not just of our system, but open source databases of documented security vulnerabilities, such as cve.mitre.org.

Perfect security is not possible. However, perfect security is not required, either. We just have to make sure that the cost to an attacker is higher than the potential reward they may gain from breaking into our system. Also, implementing some security techniques can ensure that some types of attacks are completely nullified.

The rest of the lecture presents examples on security failures to highlight the ways in which a security engineer must think. They are very interesting and fun reads, you can find them [here](#).

Lecture 2 - Security Architecture

Security architecture is structuring entire systems in order to defend against large classes of attacks, prevent as-yet unknown attacks, and limit damage from successful attacks. This lecture analyzes a paper written by Google describing their security architecture for Google Cloud Platform. It gives a good overview of the techniques and principles Google uses to implement their security architecture.

You can read the paper [here](#). You should read the paper before going through these notes. The paper describes the architecture Google uses, these notes discuss *why* Google uses that architecture and why it is good.

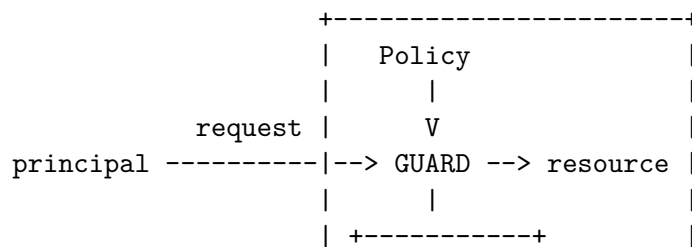
Google has the following broad security goals - 1. Protect customer data 2. Ensure availability of Google services 3. Track down what went wrong if something does go wrong 4. Limit the damage an attacker can do in case of a successful attack

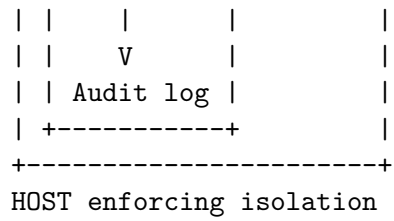
Google has the following hardware/software services in its architecture - 1. Data centers 2. Physical machines in those data centers (servers) 3. Virtual machines in those servers 4. Services running on those virtual machines 5. Applications running on those services 6. Remote Procedure Calls (RPCs) between applications and services 7. Requests coming in through the internet producing RPCs

Isolation

A fundamental principle used by Google's security architecture is isolation. Each service and/or application is isolated from each other (sometimes virtually, sometimes physically), such that by default no service can access the data and/or resources belonging to any other service. This means that even if one service is compromised or has bugs, it is not able to damage any other service. Without isolation, we cannot hope to build secure systems.

Now, we don't want every service to be 100% isolated all the time. We want services to be able to talk to each other, without which we cannot build a distributed system. We need to allow inter-service communication in a controlled way. In order to control communication between services, a common pattern is the guard pattern. A guard is placed in the front of every service, who's job is to monitor incoming requests from other services and only let them in if the service is allowed to talk to it.





In the diagram above, a principal can be a user, a device, a service, etc. Resources can be services, like Gmail or Drive, or files and user data.

The guard performs three functions -

1. Authenticate - Identifies who is issuing the request.
2. Authorize - Determines whether that request should be allowed.
3. Audit - Records each request along with its authentication and authorization information, and other useful metadata.

Lecture 3 - User Authentication

User authentication is an important problem which has a variety of issues that are hard to deal with. Even though the premise of the problem is not complicated, user authentication continues to be challenging because it is not just a technical problem. All schemes we have come up with for user authentication so far are imperfect because the end users are ultimately human and imperfect.

The bigger problem of user authentication can be broken down into three separate sub problems - 1. User registration - The process of registering the identity of a user with a service so that the identity can be verified in the future. This sets up a shared secret between the user and the service (usually a password). 2. Identity verification - The process of authenticating a user and verifying their identity. This process verifies the shared secret. 3. Recovery - The process of recovering the secret if the user loses it. If overlooked, this step can undermine the security of the first two steps, no matter how strong they are.

Identity Verification

The most common way of verifying an identity is a password. The benefit of passwords as a shared secret is that they are easy to remember and very convenient to use. However, the challenge with passwords is that the users of passwords are human, and they tend to use passwords in a way that makes them weaker, such as by reusing passwords, or by using weak or easy to guess passwords.

Passwords are never stored in plaintext. They are combined with a large random number called a salt, and hashed using a cryptographic hash function. The service then stores tuples containing (`username`, `salt`, `Hash(salt || password)`). The hash function is also purposefully made to be slow, to make it harder for attackers to brute force passwords.

However, even if passwords can be imperfect, their security can be greatly improved by augmenting them with two-factor authentication. You are already familiar with common ways of introducing two-factor authentication such as OTPs through SMS or an authentication app. These are all good approaches, but they don't protect against MITM attacks and phishing attacks.

Universal Two Factor Protocol (U2F)

The [U2F protocol](#) being developed by Google is a form of two-factor authentication which is resistant to phishing and even MITM attacks. It uses public-key encryption and a dedicated 2FA device to verify a user's identity.

It is a great improvement over OTPs over SMS or authentication apps, but it still makes some assumptions which make it vulnerable in some extremely rare situations.

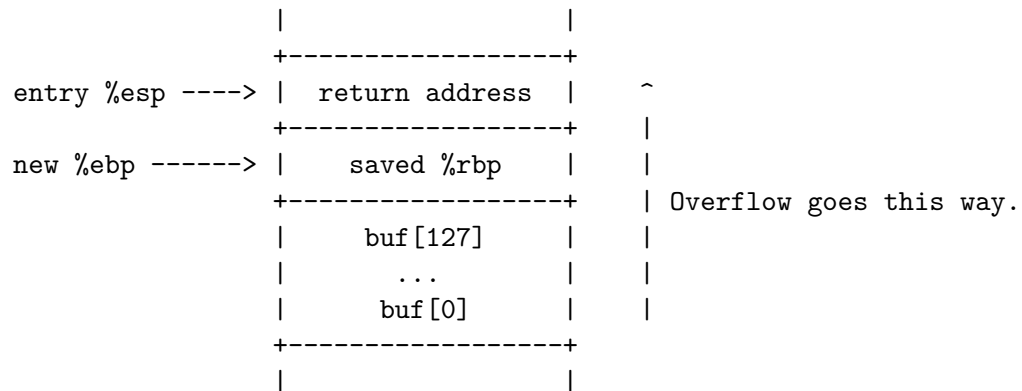
During authentication, the browser sends a login request to the server. The server sends back a “challenge”, which is a long random number, for the U2F device to sign using its private key. The user plugs in the U2F device via USB, which receives the challenge, signs it using its private key, and sends it back to the browser. The browser forwards the signed challenge to the server. The server can now verify if the U2F device’s identity using its public key.

This is the basic premise of the protocol. To protect against MITM and phishing attacks, the browser actually forwards not just the server’s challenge, but also the origin and the TLS connection ID. The U2F device then verifies the signature for all three instead of just the challenge.

Lecture 4 - Buffer Overflows

Buffer overflows are an old but still relevant attack path. A buffer overflow is an attack which uses poorly sanitised/validated user input to overwrite data on the buffer. When a function is called, the registers and the return address are stored on the stack. Sometimes, a buffer is allocated to the function if the function needs to store some data. The buffer is allocated just under the pushed registers.

If the program accepts and stores user input in the buffer and does not check the length of the input, the user input can possibly overwrite the data in the stored registers if the programming language does not check for buffer bounds before writing. For example, whenever a function is called, the stack may contain the return address, the base pointer of the stack, and finally the buffer itself. If a buffer overflow attack exists, the attacker may write data that is so long that it fills the buffer and also overwrites the base pointer of the stack and the return address.



This way, the attacker can supply any return address, which means the attacker can run any function they want. In the classic buffer overflow attack, the attacker supplies the code they want to run in the buffer itself, and the specify the return address as the first address of the buffer.

Now, this attack could be completely mitigated if the programming language simply checked if the address being written to is part of the data structure being accessed. Most modern programming languages do that. However, C doesn't. If you are not using C, then you are probably safe from buffer overflows in your code. For example, if you are using a language like Java or Rust which performs bounds checking, then you most likely won't get buffer overflows. However, you can't always control the programming language you use, and you will, at some point, have to use libraries that are written in C. Even if you don't, you will probably want to run your code on Linux, which is written in C. Therefore, it is important to learn about buffer overflows even if you don't work with C directly.

Defenses Against Buffer Overflows

The lecture takes an iterative approach to defending against buffer overflows. In computer security, we rarely have solutions that perfectly solve a problem. It is more common that we have solutions that make it harder to pull off a particular attack successfully. As we develop such solutions, attackers modify their attacks to get around these solutions, forcing us to come up with better solutions, and so on.

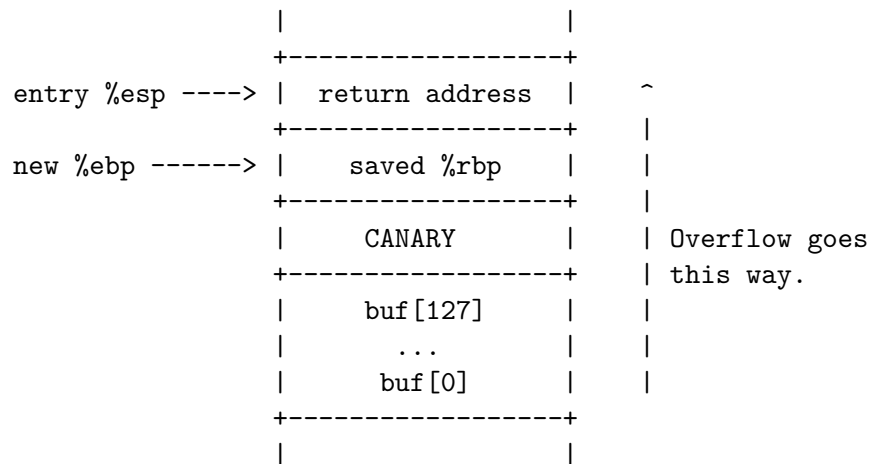
NX bit

One approach we can take is to tell the hardware not to execute any instructions in pages where the stack is stored. This is done by the operating system, since it is responsible for paging and memory management. This is a simple solution, and it definitely works, but there is a way to bypass it.

Instead of the attacker providing the code they want to run in the buffer itself, the attacker can overwrite the return address to be a `libc` function, like `exec`, and achieve the same effect of running arbitrary code. This is called a “return-to-libc” type of buffer overflow.

Stack Canaries

The next approach to protect against “return-to-libc” attacks is called a stack canary. A stack canary is a number that is stored between the stack and the return addresses and saved registers. This is a randomly generated number and is different for every process. It is generated by the operating system and stored off the stack.



When the attacker overwrites addresses outside the buffer, the canary is also overwritten. Before returning, the compiler checks the canary to make sure it is correct. If the canary is not correct, then the compiler knows that a buffer overflow has occurred and crashes the program.

This is a pretty good solution, but it isn't completely foolproof since the canary could be guessed, either by reverse engineering how the canary is generated or just by brute force.

Address Space Layout Randomization

This is a common strategy and pretty effective in combating return-to-libc attacks. Instead of storing libc binaries in the same address space everytime, we randomize the addresses where we store system libraries on every startup, as well as the location of the application stack, code, and heap. This means that the attacker cannot guess where libc functions are stored (easily), and makes it much harder to pull off return-to-libc attacks.

Lecture 5 - Privilege Separation

Privilege separation is the practice of dividing your software into multiple independent services which each run in isolation. This way one service being compromised minimizes the chance of the attacker being able to compromise another service. It also reduces the attack surface.

It is easy to describe the idea of privilege separation, but hard to actually implement it in practice. It is similar to how it is always recommended to make your software modular, but writing modular code is easier said than done.

This week's paper describes the working of a web server - OKWS - which is written with the principle of privilege separation at its core.

Recitation 1 - Linux Containers