CS3235 - Computer Security
AY25/26, Y3S1
Notes

Sim Ray En Ryan

December 3, 2025

Contents

1 Introduction **5**
- 1.1 Graded . . . 5
  - 1.1.1 Part I . . . 5
  - 1.1.2 Part II . . . 5
- 1.2 Why Computer Security . . . 5
- 1.3 Threat Models . . . 5
  - 1.3.1 Attacker's advantage . . . 6

2 Spatial Memory Errors **7**
- 2.1 Systems Programming . . . 7
- 2.2 Buffer overflows . . . 7
- 2.3 Format String Bugs . . . 7
- 2.4 Integer Overflow . . . 7
- 2.5 Control-flow Hijacking: Code Injection . . . 8
  - 2.5.1 NOP Sled . . . 8
- 2.6 Code Reuse . . . 8
- 2.7 Heap Overflows . . . 8
  - 2.7.1 Data Oriented Attacks . . . 8
- 2.8 Bad Casting . . . 8

3 Temporal Memory Errors **10**
- 3.1 Lifetime and Scope of Variables . . . 10
  - 3.1.1 Scope . . . 10
  - 3.1.2 Lifetime . . . 10
- 3.2 Use after free . . . 10
- 3.3 Double Free . . . 10

4 Mitigation Against Memory Exploits **11**
- 4.1 Defensive Coding Practices . . . 11
- 4.2 Stack Canaries . . . 11
- 4.3 Guard Page . . . 11
- 4.4 Non Executable Data / Data Execution Program . . . 11
- 4.5 Address Randomization . . . 11

5 Full Memory Safety via Rust **12**
- 5.1 Spatial Safety . . . 12
  - 5.1.1 Fat Pointers . . . 12
- 5.2 Temporal Safety . . . 12
  - 5.2.1 Lock and Key Mechanism . . . 12
- 5.3 Costs of implementing safety . . . 13
- 5.4 Smart Pointers . . . 13

5.5 Rust . . . 13
5.5.1 Spatial Safety . . . 13
5.5.2 Temporal Safety . . . 13
5.5.3 Mutability . . . 14

6 Sandboxing **15**
6.1 Reference Monitors . . . 15
6.1.1 Inline Reference Monitors . . . 15
6.2 Process Sandboxing . . . 15
6.2.1 Policies Enforcable . . . 15
6.2.2 Security Principles . . . 15
6.2.3 Privilege Separation . . . 16
6.3 Namespace Isolation . . . 16

7 Virtualization **17**
7.1 Assumptions . . . 17
7.2 Types of VMM . . . 17
7.2.1 Security VMM . . . 17
7.2.2 Commercial VMM . . . 17
7.3 Virtualization Techniques . . . 17
7.4 Hardware Assisted Virtualization . . . 18
7.5 Limitation of Virtualization . . . 18
7.6 Covert Channels . . . 19

8 Hardware Based Isolation and Trusted Execution Environment **20**
8.0.1 PKC Recap . . . 20
8.1 Remote Attestation . . . 20
8.2 Trusted Boot . . . 20
8.3 Sealing Data . . . 20
8.4 Late Launch . . . 20

9 Introduction to Cryptography **21**
9.1 History . . . 21
9.2 Encyption . . . 21
9.3 One Time Pads . . . 22
9.3.1 Proof of perfect secrecy . . . 22
9.3.2 Issue with one time pad . . . 22

10 Integrity **23**
10.1 Message Authentication Code . . . 23
10.1.1 HMAC . . . 23

11 Public Key Cryptography **24**
11.1 PKC . . . 24
11.1.1 RSA . . . 24
11.1.2 Find e . . . 25
11.1.3 Find d . . . 25
11.1.4 Example . . . 25
11.1.5 Security of RSA . . . 25
11.2 ElGamal . . . 26
11.2.1 Encryption . . . 26
11.2.2 Decryption . . . 26
11.2.3 Comparison . . . 26
11.3 Digital Signatures with PKC . . . 26

12 Secure Channels with Cryptography **28**
12.1 Key Exchange . . . 28
12.2 Diffie Hellman Key Exchange . . . 28
12.3 Summary of Secure Channel . . . 28

13 Blockchain **30**
13.1 Blockchain . . . 30
13.2 Decentralization and Proof of Work . . . 30
13.3 Consensus . . . 30
13.4 Possible Malicious Behaviors . . . 30
13.4.1 Modify Transaction Contents . . . 30
13.4.2 Deny a Transaction . . . 31
13.4.3 Double Spend . . . 31
13.5 PK Security . . . 31
13.5.1 Secret Sharing . . . 31
13.5.2 Multisig . . . 31

1 Introduction

1.1 Graded

1.1.1 Part I

- 5: Assignment on Memory Safety
- 30: Project on Rust Programming
- 15: Midterm 2 Oct 45 Minutes

1.1.2 Part II

- 30: Finals 12 Nov
- 20: Team Project

1.2 Why Computer Security

- Has actual threat to lives : Attacks on infrastructure, electricity
- Underground economy
- Data Theft
- Number of vulnerabilities and attacks increasing exponentially

1.3 Threat Models

For example, signing a credit card:

- Setup: The real owner must have signed the card
- Assumption: Merchant asks presenter to physically reproduce signature
- Attack Capability: Unable to forge physical signature or collude with merchant
- Attack Goal: Impersonate and profit

However, nobody signs the cards, merchants never do that, attackers can easily forge physical signatures, and they still can profit. Nothing is ever secure, and we will analyze the cost of attackers and defenders at each step to find the balance.

1.3.1 Attacker's advantage

Certain attacks can be very targeted, but costly to repeat at scale. Others may be cheap for attacker to repeat at scale.

Hardware exploits are difficult for defenders to patch, but easy to exploit once discovered for attackers. Malwares can be solved easily but also deployed easily. Captchas are an easy way for defenders to prevent spam.

2 Spatial Memory Errors

Browser and server code are often written in C and CPP, which have very fine control over memory but can be memory unsafe. Programmers can write code to fix problems but may make mistakes that can be exploited.

2.1 Systems Programming

x86 has an instruction pointer that will point to the current assembly code that should be executing. In x86, the stack grows down and the heap grows up Since both code and data are numbers. A stack frame is used for the stack (as learnt in CS2106)

2.2 Buffer overflows

For example, f calls g. g defines a buffer of 50 characters and the stack frame will allocate a space of 50 bytes to contain the characters. Since scanf does not define a maximum, placing more than 50 bytes will allow you to rewrite what is after that allocated space, which may contain the return address.

2.3 Format String Bugs

If the programmer does not explicitly provide all arguments to a format stirng, printf will still think it has all arguments. Then, it will read what should have been there, which can contain sensitive information.

- %d: Number
- %p: Pointer
- %c: Character
- %s: Read address, print bytes until NULL byte (the most dangerous, it may just print out the whole stack until a NULL happens to be there)
- %n: Write number of bytes already printed (allows printf to write to memory)
- n$: Access nth positional argument

2.4 Integer Overflow

A 32 byte buffer is set, and you have strlen to check the input length. Then a conditional to check if the len is less than 32. This may not work properly: If 8 bit numbers are used, you just need to write 255+ or 127+ characters to get a negative or number below 32, then allowing you to write into another buffer anyway.

2.5 Control-flow Hijacking: Code Injection

For example, this program is running in privileged access. Once an attacker writes an attack payload in memory, change that payload to be executable, they must then divert control flow to that payload.

2.5.1 NOP Sled

You add pages and pages of NOP so that the chances of jumping to a random address will eventually lead you to the shellcode. Some antivirus will detect this.

2.6 Code Reuse

A stronger version of code injection, where an attacker jumps the programme directly to an attackers controlled code address. For example, execve is usually linked and in the memory of most programs. Since execve also just reads from stack, we can attempt to jump to the address and make it execute using our own stack arguments to get any other program.

Return Oriented Programming is also a type of code reuse attack, which is more general than return-to-libc attacks which is more general than return to execve attacks.

2.7 Heap Overflows

Nothing above is specific to stack. Exact same happens to heap when you malloc(x) and write more than x bytes. In the case of C, overwriting will also overwrite the pointers in a doubly-linked list and corrupt it. This is very good for data oriented attacks. For example, if there is a function that de-escalates privileges. Corrupting or NOPing it will allow a user to remain with root.

2.7.1 Data Oriented Attacks

Do not need to corrupt or rewrite anything for malicious outcomes. Heartbleed bug in OpenSSL allowed one to leak data.

2.8 Bad Casting

Cpp has support for OOP. When you have a pointer to a derived object, it usually also contains a pointer to the base object at the beginning. By inheritance, it is ok to upcast a derived type to a base type.

In terms of pointers, they will still be pointing to the base object at the start, which means that upcasting is always safe. Even with multiple inheritance, Cpp will point it to the correct beginning of the second base class (by moving it the size of the first base class)

It is also possible to downcast, although it may not have specific properties that the derived classes have. When you actually downcast, Cpp will assume that the object IS

already of that type, and just use the data as it expects. For example, if you have down-casted to the second base class, they will see the expected data in the first base class, leading to leaking, crashing, or writing beyond the bounds of memory. This can happen even with single inheritance if the object itself is not the derived class.

3 Temporal Memory Errors

In execution of a program, the same space of memory contains different objects (due to malloc and free) at different points in time.

A temporal memory error is when a program accesses memory beyond its valid lifetime.

3.1 Lifetime and Scope of Variables

3.1.1 Scope

Region of code where a variable can be accessed. This can be a global scope, function-local scope, or in heap.

3.1.2 Lifetime

The period of exection time in which the object is guaranteed to be in memory. This also includes the local variables of f when f calls g. g cannot access these variables but they will still be alive

3.2 Use after free

After an object has been freed or deallocated, it MAY still contain its original values but typically has undefined behavior as any other thread or process can now write over it. If you still reference the object after it has been deleted, then it is a temporal error of being used after its valid lifetime.

3.3 Double Free

```
1. char* p, q;
2. p = (char *) malloc(SIZE);
3. if (abrt) free(p);
4. q = (char *) malloc(2 * SIZE);
5. strncpy(q, ext_input, 2*SIZE);
6. free(p);
```

After line 3, if p is freed, then alocating q in line 4 will likely have the same starting address as p. Then, free(p) on line 6 will access a variable outside of its lifetime.

Assuming p is a doubly linked list, then calling free(p) on data that is not a linked list will lead to unexpected behavior. In this case

```
if (p->next) p->next->prev = p->prev
```

Since in line 5, you can write q to be any external input, you can specify any part of the memory (p next prev) to contain anything (p prev)

4 Mitigation Against Memory Exploits

4.1 Defensive Coding Practices

- Use bug finding tools: Static Analyzers (address sanitization, ITS4, RATS, flawfinder) or fuzzers to find errors
- Safe Coding Techniques. In particular loops, and explicitly specifying the size of destination buffers (strcpy and strncpy) (helps with spatial memory safety)
- Using safe libraries. Libsafe checks for uses of strcpy explicitly

4.2 Stack Canaries

Add a secret data value to the stack right before the return address. If the canary still has the same value after the function, then it is an indication that a buffer overflow has not happened that affects outside the function.

4.3 Guard Page

Mark certain pages with not readable, writable, or executable. These are placed randomly in the memory, and we assume that the attacker can only write to memory linearly. When one of these pages get triggered, the program will shut down.

4.4 Non Executable Data / Data Execution Program

Many exploits require both writable and executable. This program makes sure that W or X, but not both. This prevents foreign code injection and blocks requirement 2 of the attack. Of course, this does not prevent leaking attacks, and many programs also need parts of code to be both writable and readable. E.g a JIT compilers.

4.5 Address Randomization

Many exploits require knowing roughly where the target is. Although an attacker can write to arbitrary locations in memory, it cannot predict the location accessed in the attack. Address Space Layout Randomization ASLR randomizes the stack, code as load time. (offset stack)

5 Full Memory Safety via Rust

Programmers can create memory pointers via permitted operations, and memory is only accessed when it is within the allocated range and when the memory is in scope. All objects are sptially disjoint at all times. These 3 conditions can be enforced via specific compilation or by binary rewriting (decompile, add certian instructions, recompile), or by inserting metadata and inline monitors.

5.1 Spatial Safety

Associate bounds with each pointer or object, and we check each pointer before access. Since pointers can be operated on, this needs to be kept in mind as well. A fat pointer contains the start and end address it is allowed to access, and a referent object does the same but within an object. Some systems may break since they cannot support these, and a solution is to have the addresses stored in some other table.

An assumption is that the metadata cannot be corrupted.

5.1.1 Fat Pointers

On allocation, track the bounds:

```
ptr = malloc(size); // int array[100];
ptr_base = ptr; // ptr = &array;
ptr_bound = ptr + size; // ptr_base = &array[0] # points to same location but diff type
if (ptr == NULL) ptr_bound = NULL; // ptr_bound = &array[100];
```

When you dereference a pointer, do a check. `check(ptr, ptr_base, ptr_bound, sizeof(*ptr))` and if the value is outside of the two bounds, abort the operation.

When there is pointer arithmetic, unsafe typecasts, do not need to update bounds as invalid accesses will immediately be caught and aborted. For safe type casts, have to update pointer bounds to be smaller (it is also ok to keep it the same bounds as it allows for memory safe downcast later).

5.2 Temporal Safety

NULL-ing a pointer is ok. But when multiple pointers point to the same location, it is hard to keep track of all of them.

5.2.1 Lock and Key Mechanism

Tracks creation and destruction of pointers and ensures that deallocated pointers are not accessed. Objects have a 'key' to a memory location and a memory location has a 'lock' that changes each time a new object is referenced or dereferenced. In order to access

the location, the two values must match, meaning that it IS the current object and is temporally safe.

In practice, a pointer will have the key, and a pointer to the lock address which contains a second key. This only works under the assumption that these values cannot be corrupted.

On deallocation, change the local value to an invalid address, such that no other object can wish to use it.

5.3 Costs of implementing safety

Overhead in having extra memory to store bounds, time in accessing these memories which now cannot exploit spatial locality as much.

Overhead in having temporal safety is larger. The average perfoamnce overhead is about 200%

5.4 Smart Pointers

A smart pointer owns a resource object and frees the object when pointer goes out of scope. Smart pointers are unique, and nothing else can point / it transfers ownership (moves). This prevents dangling pointers and avoids temporal errors.

This is restrictive since some programs need more than one pointer to point to the same location. Shared pointer smart pointers track how many pointers point to the resource, and only free when the count goes to 0. This however now results in runtime overhead.

5.5 Rust

5.5.1 Spatial Safety

Rust uses fat pointers to ensure spatial safety

5.5.2 Temporal Safety

There is an exclusive owner of every object at any given point in time. It cannot be duplicated but only moved. A referent object is auto destroyed when an owner destroys it. Thus, variables may not be used again once ownership is passed.

Local and Global variables are known at compile time. Heap objects are owned by local or global variables, and heap objects can then own other heap variables.

Borrowing without ownership transfer By using ampersand, you can borrow a variable, and you will only use the alias of the object. At the end of the lifetime, this object will not be deallocated. Borrowing is only okay when it does not live past the owner.

5.5.3 Mutability

There are immutable and mutable references in Rust. Immutable can only be read, while mutable can read and write. Variables are immutable by default. The borrower also accepts immutable by default

```
let mut x = 5;
{
  let y = &mut x; // y is a mutable reference
  *y += 1;
} // scope of y ends
println!("{}", x);
```

At each line, there is one unique mutable reference. Once an object is mutable, it cannot be borrowed as immutable. The lifetime of objects cannot be longer than their referents.

```
let mut x = String::from("Singapore");
let x_ref = &x;
println!("{}, x_ref);
let y = func(x); // trying to use x when x_ref still exists
println!("{}", x_ref):
```

TODO.

6 Sandboxing

A second line of defense. The first line directly prevents the attack. The second line assumes that the attack happens, and minimizes the impact.

6.1 Reference Monitors

A piece of code that checks all references to an object. A syscall sandbox is a reference monitor for protecting OS resource objects from an app, and is located within the kernel

A reference monitor will

- Check if the user has enough access rights to get the object. (Ref Monitors may not do the enforcement)

6.1.1 Inline Reference Monitors

Is written within the code itself and above the kernel.

6.2 Process Sandboxing

Syscall policies such as no exec after system call, or no exec after read.

6.2.1 Policies Enforcable

Linux seccomp cannot make any syscalls except exit(), sigreturn(), read(), and write().

Linux seccomp-bpf is more configurable but more complex

Linux security models. Policies can include syscall data arguments as well

Allow listing is better than block-listing

6.2.2 Security Principles

- Separation of Concerns: Separate the policy from enforcement
- Minimize Trusted Code Base: Reduce what one needs to trust and separate the verifier from the enforcement. A TCB is what is required that needs to be guaranteed to be safe for every other argument to hold.
- Least Privilege: Each component only gets the privileges necessary

6.2.3 Privilege Separation

The SSH server should only deal with network logic. If it fails, at least it will not affect the filesystem. Each compartment gets the least set of privileges it needs for its functions.

- The network sandbox will only have access to OS network functions, but not read or write
- The filesystem sandbox will only have access to OS file functions, but not send or receive

Web Browsers used to have a huge codebase, many languages, and handle everything. The ways to fix this was auto-patching of old bad code. Chrome also had separation of filesystem from the web code to prevent websites from downloading things into your computer.

- Rendering Engine: HTML, CSS, Image, JS, Regex, Layout, DOCM, Rendering, SVG, XML, XSLT
- Browser Kernel: Cookies, History, Password, Windows, Location, Network Stack, SSL, Disk, Downloads, Clipboard
- Both still handle URL parsing and unicode parsing

6.3 Namespace Isolation

The namespaces of procesess are different. Thus they can never see and then affect each other.

7 Virtualization

On the assumption that softwares are no longer just 'benign but bugggy', but are malicious softwares explicitly written to bypass your defenses.

Virtualization helps with isolation. If there are multiple OS on the same CPU, both OS can mess with each other. Instead, make a virtual machine monitor run on the hardware, and have each VM run on the VMM. The VMs can be split by importance also such as a Banking VM and a VM for pirating games. This is an example of red-green systems.

VMs can also be used for dynamic analysis of malware, as well as its containment. Running an antivirus in the VMM is called Virtual machine introspection and allows us to observe the guest OS closely.

7.1 Assumptions

Our goal is to have isolation of code, data, and resources between guest VM and host VMM, as well as between VMs. Our assumptions are that

- The Trust Code Base: Host OS and VMM are bug free
- That malware can only affect the guest OS and apps.

7.2 Types of VMM

7.2.1 Security VMM

allows for complete mediation, traps on all MMU, DMA, and IO accesses, and gives transparency of the gues OS

7.2.2 Commercial VMM

allows for higher performance, and compatibility by running on commodity OSes.

7.3 Virtualization Techniques

Binary Translation is used by VMWare, where the VMM scans for instructions that needs to trap, then replaces it with a trap that will go to the hypervisor emulator.

Paravirtualization is used on Xen, and it modifies the guest OS to directly call the hypervisor to access physical resources

7.4 Hardware Assisted Virtualization

CPUs have been adding support to improve performance and security.

- MMU virtualization using EPT
- Nested virtualiztion using VMCS
- IO virtualization using IOMMU
- DMA Remapping

7.5 Limitation of Virtualization

It is useless if the attacker is located at a higher (lower really) level than the VMM (if the VMM is running on a bad VMM).

Malwares can also detect if they are being virtualized:

- Checking System Manufacturer or BIOS version for strings such as VMWare
- List devices and look for VM vendors
- Check for installed tools such as VMWare Tools
- Registry keys of VMWare
- Environment variables containing VM Vendors
- The hypervisor will set a hypervisor present bit
- Hypervisor Vendor ID can be returned with another function ID (?)
- Low level instructions interacting with virtualization layer takes much longer to execute. Time this execution.
- Check resources limits, or performance limits as VMs usually have less.
- Screen resolution, color dpeth, refresh rate
- Lack of user activity / installed programmes.
- Sometimes VMMs emulate old chipsets

Malicious software can then not execute to prevent inspection or execute with anti-antivirus methods. Actual software can also use this to prevent itself from being copied. (VMs can be duplicated resulting in multiple copies of the same licensed softwares)

7.6 Covert Channels

An unintended channel of communication between 2 untrsuted programs. For example, through shared cache latency and check if cache is available (DNS caching attack). This attacks can get 0.02 bits per second. VMs are good for integrity as they will not be able to corrupt data outside the VM, but it is not confidential as things can be leaked out.

8 Hardware Based Isolation and Trusted Execution Environment

Since the security of a model must depend on the how much the lowest level is trusted, the simplest way to ensure secure root of trust is to make the HARDWARE trustable and assume that all software is malicious. Hardware cannot be changed by software malware, and is easier to verify that it is safe.

- Confidentiality is achieved by encryption
- Integrity (including authentication) is achieved by MAC and Digital Signatures

8.0.1 PKC Recap

Public keys are distributed via secure channel. You can sign a message with your private key and anyone can verify with your public key. Anyone can encrypt with your public key and you can decrypt with your private key.

8.1 Remote Attestation

For all software stack layers below the application VM that needs to be verified, hash them (extend the hash into a PCR 1 .. n), creating a unique fingerprint of the entire boot sequence and configuration. and then sign it with the verifiers (Attestation Identifier Key) key. Each Trusted Platform Module has a AIK signing key to ensure that it has not been tampered with.

8.2 Trusted Boot

A static root of trust measurement system performs checks and verification and load time, checking if the initial software components such as the BIOS and boot loader are in a trustworthy state before loading the OS.

8.3 Sealing Data

The Disk is encrypted by the TPM that can only be accessed if the current PCR values match the values present when the data was sealed. Full Volume Encryption such as by Windows Bitlocker uses this.

8.4 Late Launch

Dynamic Root of Trust Measurement allows trusted environments to be launched later in the systems execution. The TPM hashes a specified memory region into a PCR. If the hash is valid, it will then execute the code if the hash matches. These operations must be done atomically. ??? TODO

9 Introduction to Cryptography

Cryptography = Encryption + Decryption. This allows for the confidentiality to pass information secretly.

9.1 History

- Caeser Cipher: Shift all alphabets 3 places forward (A to D). Is just encoding
- Keyed Cipher: Keep secret k, replace shift by k. Only need 26 tries to determine if any data is meaningful
- Substitution Cipher: Bijection from alphabet to alphabet. Weak to frequency attack

Unlike classic cryptography, modern cryptography has:

- Formal Definitions based on mathematical models and what security means
- Assumptions about each model
- Proof of security based on definition and assumptions.

9.2 Encyption

Is defined as:

- KeySetup(): Outputs the private key k
- Enc(m, k): Encrypts message m with k to output c
- De(c, k): Decrypts ciphertext c with k to output m

Encryption must be correct: Dec(Enc(m, k), k) = m, and it must be secret:

- Adversary cannot recover m given c
- Adversary cannot find k
- Attacker cannot determine any information about m given c. Informally, people who have access to c do not have any advantage over those who don't.

More definitions:

- K: Key space: The set of all possible keys. K should have a uniform distribution.
- M: Message space: The set of all possible plaintext.
- C: Cipher space: The set of all possible ciphertext

Formally, perfect secrecy is where $\forall m \in M, c \in C, Pr[M = m|C = c] = Pr[M = m]$, or that the distribution of M does not change even when conditioned on observing the ciphertext. Correctness is $\forall m \in M, k \in K, Dec(Enc(m, k), k) = m$

9.3 One Time Pads

XOR is commutative, associative, has identity 0 and is a self inverse.

One time pad works by first generating an n-bit private key. With an n-bit plaintext, bitwise XOR to obtain ciphertext, and bitwise XOR again to obtain plaintext.

9.3.1 Proof of perfect secrecy

Have an abritrary message space $M = \{0,1\}^n$, same for m and c.

$Pr[M = m|C = c] = Pr[C = c|M = m] \cdot Pr[M = m]/Pr[C = c]$

TODO

Then, each ciphertext has a uniform distribution. Given a c = 0 on the cipher text, x can be either 0 or 1,thus it is perfectly secret.

9.3.2 Issue with one time pad

Key space is at least as large as the message space, and it is necessary to satisfy perfect secrecy. However, it is not feasible in everyday use, and PERFECT secrecy is not needed.

If message space is smaller than ciphertext space, then two messages will have to be mapped to the same ciphertext. Since this is impossible (by correctness property), that must mean that the ciphertext space is at most the size of the keyspace. If k is less than c, then not all ciphertexts can be mapped. Then there is an invalid ciphertext that cannot exist therefore violating perfect secrecy.

One time use If $c = m \oplus k, c' = m' \oplus k$ then $c \oplus c' = m \oplus m'$. This reveals information about the plaintext. Furthemore, as it is very large you have to send the key many times

Kerchkhoff's Principle A cryptosystem should be secure, even if everything about the system is known. Thus, the method to create a key, encrypt, and decrypt can be known, an the only secret should be the key. It is easier to keep the key secret than the algorithm, and it is easier to change the key. It is also better for the algorithm to be known so that it can be validated by others and used widely.

10 Integrity

Alice can send a message to Bob. Even with encryption, Mallory can change c to c' and possibly change the value of the message. Bob has no way of knowing that c has been changed.

10.1 Message Authentication Code

Ensures the sender authenticity and any changes to the sent message are detected during verification.

Hash Function Takes an arbitrarily large message and outputs a fixed size digest. Although there must be collisions, hash functions should be:

- Collision Resistant: Hard to find x and x' such that H(x) = H(x') but x != x'.
- Not inversible, given digest, hard to find x such that H(x) = d. This is safe under the assumption that computational power cannot brute force this.

Hash functions are used in passwords. Storing hashes makes weak passwords look longer, less recognizable, password is not leaked when password file is leaked.

Hash functions are also used for software integrity. Publishing the shorter hash on a secure broadcast channel, allow users to compare. It is almost impossible for a malicious actor to modify the base file while still having the same hash.

10.1.1 HMAC

Alice adds a short hash H(m, k) to the end of the message. k is the private key of Alice that Bob can check with her public key. Without the k, anyone can generate a new hash. Ohterwise, now Mallory needs to construct a new tag H(m', k) without knowing k. MACs do not have perfect secrecy, but have computationally secrecy.

Perfect MAC An adversary has no advantage (still 2^{-n}, a random guess) in forging a valid tag even after seeing the tag.

$$P(tag(m,k) = t \cap tag(m',k) = t') = \frac{1}{|T|^2}$$

The keys should be at least 2n for an nbit perfectly secure MAC.

TODO. WHAT???

Perfect security has impractical key sizes. For MACs, the key space size must be more than the square of the tag length.

11 Public Key Cryptography

A symmetric key encryption uses the same key for encryption and decryption. PKC uses a different key for encryption and decryption. In encryption, encrypt with other side's public key. In decryption, decrypt with your own private key.

PKC achieves confidentiality without shared secrets. This is useful for open environments, and can use it to establish keys for symmetric encryption. It is easy to use with multiple parties. However, it is 2 to 3 times slower than symmmetric encryption, and relies on unproven number theory assumptions.

11.1 PKC

Different from asymmetric has the public key is public to all, even adversaries. PKC significantly reduces the number of keys from a complete graph (n(n-1)/2) to just one pair of keys per user.

Correctness Dec(Enc(m, pk), vk) = m

Secrecy Given the public key and ciphertext, difficult to determine plaintext

Symmetric keys also require both entities to know each other before the actual session, while PKC only needs a secure broadcast channel to distribute public keys. PKC is also more vulnerable to quantum computing.

11.1.1 RSA

- Choose 2 large primes p, q so that n = pq
- Choose random encryption exponent e such that gcd(e, (p-1)(q-1)) = 1 (no common factors)
- Choose decryption exponent d such that de mod (p-1)(q-1) = 1.
- Easy to calculate d given e, p, and q
- (p-1)(q-1) is the Euler Toitent function, which is ther number of coprimes under n.
- Publish n and e as the public key. P and Q are not required. Keep d as the private key
- Security relies on the fact that pq = n but difficult to find n = pq.

Some complicated math

$m^{\phi(n)} mod n = 1 mod\, m = c^d mod n = m^{ed} mod n = m \cdot (m^{\phi(n)})^k mod n = m mod n\; ed mod \phi(n) = 1\; ed = k\phi(n) + 1$

11.1.2 Find e

Choose e as a prime such as 3 or 65537. Can just regenerate e, p, or q until condition is satisfied. Does not need to be complicated as e is public.

11.1.3 Find d

Use Extended Euclidean Algorithm given e, p, and q to find d.

For example,

- p = 5, q = 11, e = 3
- then n = 55 and $\phi(n) = 40$.
- gcd(e, (p-1)(q-1)) = 1 (fulfilled)
- de mod (p-1)(q-1) = 1
- So 3d mod 40 = 1 == 3d + 40k = 1
- 40 = 3 * 13 + 1
- 3 * -13 + 40 = 1
- so let d = -13 = 27 under mod 40.

11.1.4 Example

- Let m = 9
- Then c = $m^e mod n = 9^3 mod 55 = 14$
- Then d = $c^d mod n = 14^{27} mod 55 = 9$

11.1.5 Security of RSA

Problem of getting RSA private key from public key is as difficult as factoring n. Unknown if problem of getting plaintext from ciphertext is as difficult as factorization.

It is not secure under an unbound adversary, so it cannot be perfectly secret. 829 bits was cracked in 2020. In real world deployments, we are using at least 2000 bits.

11.2 ElGamal

Based on the assumption of the discrete log problem.

For large prime p and $g < p$, hard to find x such that $y = g^x \text{mod } p$ where y is not 1.

To decrypt, adversary needs to compute c_1^x, and then attacker needs to compute discrete log.

- Choose large prime p and g
- Pick random x as private key, $1 \leq x \leq p - 2$
- Compute public key $y = g^x$ mod p
- The public key is p, g, and y

11.2.1 Encryption

- Choose random k, $1 \leq k \leq p - 2$
- $c_1 = g^k \text{mod p}$
- $c_2 = m \cdot y^k$ mod p
- C = c_1, c_2

11.2.2 Decryption

Use c1 to get k, and use k to recover m

- $s = c_1^x \text{mod p}$
- Find s^{-1} the multiplicative inverse of s
- m = $c_2 \cdot s^{-1} \text{mod p}$

11.2.3 Comparison

RSA is deterministic, while ElGamal is probabilisitic based on k. Both are susceptible to quantum attacks, and both have a ciphertext size TODO. RSA has fast encryption with small e, but slower decryption. ElGamal is slower.

11.3 Digital Signatures with PKC

To compute a signature. must know private key. To validate signature, must know public key. Use the private key to "encrypt" the message. It is difficult to forge a signature without knowing

Existence Forgery Attack If you know the signature of m and m', then due to homomorphic property, it is easy to create signature of mm'. To solve this, hash the message first then decrypt it. To validate, decrypt using public key, and then check if hash matches. Hash is good as it gives a fixed length, makes the message to be validated shorter as well.

12 Secure Channels with Cryptography

With the above, can obtain a secure (authentic and confidential) channel on top of an underlying unsecure public channel. Confidentiality is achieved by encryption, integrity by MAC and signature, and Authentication by signatures and MAC.

To combine, do MAC on the encrypted ciphertext. If doing both separately, each part is working individually and are decoupled. If doing mac then encryption, you cannot guarantee that the message has not changed.

12.1 Key Exchange

With PKC, each PKC will create a new session. On each session, Alice and Bob will now communicatr with this new key. In the real world, the private key may eventually leak: using session keys makes sure all communications remain private unless that specific private key is leaked.

Session key should be destroyed after each session.

Forward Secrecy Even if long term private key is compromised. past session keys are secret and an attack cannot decrypt previously recorded communications. (The ciphertexts still can be stored)

12.2 Diffie Hellman Key Exchange

- Alice generates secret a and Bob generates secret b
- Both agree on a generator g
- Both send g^a and g^b over.
- Both use a and b to calculate $(g^b)^a = (g^a)^b = k$ which will be the session key.
- Eve cannot learn the session key (Computational DH is harder than discrete log)

Diffie Hellman is not secure with Mallory. Mallory can MiTM. Mallory chooses their own secret value c. Mallory intercepts the communication and sends g^c to both Alice and bob, creating two session keys for the same session. Can be solved using integrity: signing g^a, g^b. This is known as Station-To-Station Protocol.

12.3 Summary of Secure Channel

- Alice wants to visit https://bob.com in the presence of Mallory
- Alice obtains Bobs public key

- Alice and Bob carry out authenticated key exchange protocol. Alice now knows that Bob is authenticated.
- Alice now uses the session keys to communicate with Bob

13 Blockchain

Specific application of cryptography, creates trust as the basis of the economy. Must trust the server in transactions beyweem clients, but how do we know to trust the server. Blockchains provides coordination between many parties in a virtual world when there is no single trusted party.

The security of the blockchain ensures the health of the mining ecosystem ensures the value of the currency ensures the security of the blockchain.

13.1 Blockchain

Time ordered linked ledger of transactions visible to everyone and impossible to tamper with. Everyone has an address (public key), and a private key. Each block has a hash pointer to link transactions and blocks. Hashes are used as it will detect when someone modifies the data, and has a small digest.

13.2 Decentralization and Proof of Work

If a central authorithy is malicious, they can modify any transaction. With decentralization, thousand of miners will reach a consensus that the block is legit. Mining a block is a proof of work as it finds the hash that has low probability. The successful property is recalculated every two weeks as computing power changes. Reisitant against sybil attack as the probability that eve can win the next block is the fraction of global power controlled, and not the number of accounts. The nonce published in the block is very trivial to verify.

13.3 Consensus

Malicious miners can create conflicting blocks and create a souble spending attack. In blockchain, the nodes must agree on which transactions are recorded, the order of trasnactions, and create a single chain accepted by the entire network.

Transactions are compiled into candidate blocks and solved to get a Proof of Work. Then the blocks are chained. Newly minted bitcoins are given to the minors of a valid block, and payments from users included in the block are given. Only the longest valid change is valid, and miners are incentivised to follow the rules. This results in an implicit consesus without explicit voting.

With 6 confirmations, it is impossible to reverse a transaction as long as no one controls more than 50% of total hash power.

13.4 Possible Malicious Behaviors

13.4.1 Modify Transaction Contents

Guaranteed by cryptography. Malicious miner does not know other addresses secret keys, and cannot change the contents without affecting the signature.

13.4.2 Deny a Transaction

Every honest miner will include the transaction. Only if 51% of miners are malicious.

13.4.3 Double Spend

Multiple rounds of confirmation. Global consensus will be stable after several rounds of confirmations.

13.5 PK Security

PK is the ownership of coins. If leaked, anyone can spend your coins. If lost, coins are gone forever. Keys need both security and availability. This includes wallet softwares, mnemonic ohrases, hardware wallets, secret sharing, and multisig.

13.5.1 Secret Sharing

Create a polynomail where the secret is f(-1). With f(1) and f(2), you cannot reover the polynomial, but with f(3) you can.

13.5.2 Multisig

Three of four keys must be used to release a coin.