CS 161 Lecture Notes

Mokhalad Aljuboori

Fall 2023 – Inst. Peyrin Kao

08/23/23

Lecture 1

Introduction and Security Principles

1.1 Security Principles

1.1.1 Know your threat model

Definition 1.1: Thread Model

A model of who your attacker is and what resources they have.

Always assume the attacker can interact with systems without notice. Knows general information about systems (OS, vulnerabilities in software, etc). Assume the attacker is persistent and lucky.

1.1.2 Consider Human Factors

It all comes down to people.

For the users, if a security system is hard to use, users will find a way to subvert that system, compromising their system to make their lives easier.

For the programmers, they are likely to make mistakes, which are more frequent with some programming tools that others (C and C++ users make a lot of mistakes).

1.1.3 Security is Economics

The stronger the security, the more costly it will be. Therefore there should be a cost/benefit analyses to determine the security level we need. The expected benefit of your defense should be proportional to the expected cost of attack.

1.1.4 Detect if you Can't Prevent

TODO

- **Deterrence**: Stop the attack before it happens
- **Prevention**:
- **Detection**:
- **Response**:

1.1.5 Detection in Depth

An attacker should have to breach all defenses to successfully attack a system; Therefore, layer multiple types of defenses.

1.1.6 Least Privilege

Grant the minimum amount of access to applications/users needed to preform the tasks they need to do.

1.1.7 Separation of Responsibility

If you need to have a privilege, consider requiring multiple parties to work together to exercise it.

Example 1.2: Movie Theater Workers

Why is there two workers at the theaters that handle ticketing? One to give you the ticket and one to cut the ticket? This is a form of Separation of Responsibility to prevent insiders fraud.

1.1.8 Shannon's Maxim

Assume that the attackers knows your system. You should never reply on obscurity as part of your security. Always assume that the attacker knows every detail about the system you are working with (algorithms, hardware, defenses, etc.)

Design in Security from the Start

08/24/23

Lecture 2

x86 Assembly and Call Stack

2.1 Terminology

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

A "word" is the size of a pointer, which depends on your CPU architecture.

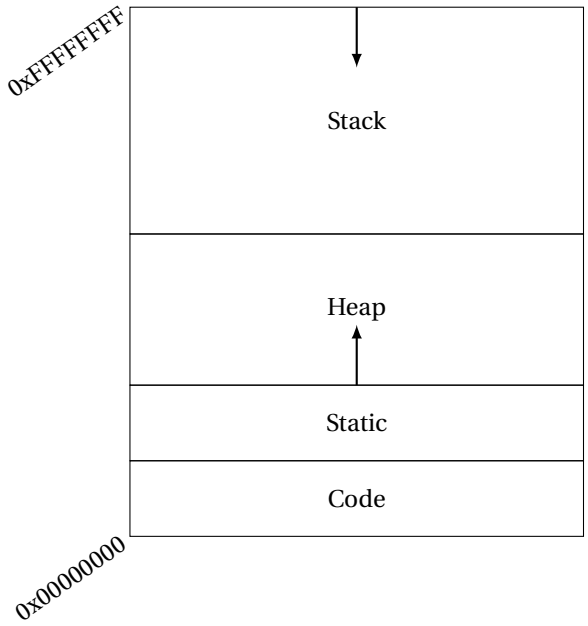
2.2 Compiler, Assembler, Linker, Loader (CALL)

There are four main steps in running a C program.

1. The compiler translates your C code into assembly instruction.
2. The Assembler translates the assembly instructions from the compiler into machine code.
3. The linker resolves dependencies on external libraries. After the linker is finished linking external libraries, it outputs a binary executable of the program that you can run.
4. When the user runs the executable, the loaders sets up an address space in memory and runs the machine code instructions in the executable.

2.3 C Memory Layout

At runtime, the OS gives the program an address space to stare any state necessary for program execution. Each byte of memory has a unique address.



- The stack stores local variables and other information associated with function calls. The stack starts at the highest address and "grows down."
- The heap is dynamically allocated memory using `malloc` and freed with `free`.As more and more memory is allocated, it grows upwards.
- The Data is where static variables, which are allocated when the program is started.
- The Code is where the program code (machine code) lies, also called "text".

2.4 Registers

In addition to the 2^{32} bytes of memory in the address space, there are also registers, which store memory directly on the CPU where Each register can store on word. There are three special x86 registers that are relevant for these notes.

- *eip* is the instruction pointer, and it stores the address of the machine instruction currently being executed. In RISC-V this register is called the PC.
- *ebp* is the base pointer, and it stores the address of the top of the current stack frame.
- *esp* is the stack pointer, and it stores the address of the bottom of the current stack frame. In RISC-V, this register is called the SP (stack pointer)

Question: Why is there an e at the start of these register name?
The e stands for "extended" and indicates that we are using a 32-bit system (extended from the original 16-bit system).

Question: why do we need a base pointer if the stack always start at the highest address 0xFF...
base pointers change as you make function calls, which open new frames with it's own local variables.

2.5 Stack: Pushing and popping

The x86 `push` instruction allocate additional space on the stack then store the value in the newly allocated space.

The x86 `pop` instruction increment the *esp* so that the popped value is now below the esp. *Note that the value is not wiped away from memory, it will be below the esp, which is now in undefined memory.*

2.6 Calling Convention

This class uses the AT&T x86 syntax. This means that the destination register comes last.

References to registers are preceded with a percent sign.

Immediates are preceded with a dollar sign (i.e. $1, $0x4, etc).

Memory references use parenthesis and can have immediate offsets; for example 12(%esp) dereferences memory 12 bytes away above the address contained in ESP.

2.7 x86 Function Calls

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

- eip, the instruction pointer, is currently pointing at the instruction of the caller. It needs to be changed to point to the instruction of the callee.
- ebp and esp currently point to the top and bottom of the caller stack frame, respectively.

When returning from a function, the *ESP*, *EBP*, and *EIP* must return to their old values.

Here are the steps of calling a function.

1. Push the arguments onto the stack in reverse order.
2. Push the current value of eip on the stack, this value is called rip of the callee, which stands for return instruction pointer
3. Push the EBP on the stack. This value is sometimes called sfp of the callee which stands for saved frame pointer.
4. TODO...
5. Begin function execution

Question: if ebp is moved below the arguments, how can we have access to them in the called function?
Question: which function calls main? Do we save the rip and sfp of main?

08/30/23

Lecture 3

Memory Safety Vulnerabilities

3.1 Buffer Overflow Error

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

Example 3.1: Buffer Overflow Attack

Here is a piece of code that uses the gets function. The gets function will write bytes until the input contains a newline (\n), not when the end of the array is reached.

```
1 char buf[8];
2 int authenticated = 0;
3 void vulnerable() {
4     gets(buf);
5 }
```

In C, static memory is filled in the order that variables are defined, so authenticated is at a higher address in memory than buf.

If the user units something like: 12345678**?** This will override what the variable `authenticated` stores into whatever we want, allowing us to maybe have privileges we weren't suppose to.

In general the user can override any variable that's defined under the variable inside the gets function (whatever gets is writing into.)

3.2 Stack Smashing

The most common kind of buffer overflow occurs on stack memory. What are some values on the stack an attacker can overflow?

- Local variables
- function arguments
- sfp
- rip

Recall when returning from a program, the EIP is set to the value of the RIP saved on the stack in memory. Stack smashing can be used for malicious code injection.

Example 3.2: Overwriting the RIP

Consider this program:

```
void vulnerable(void) {
    char name[20];
    gets(name);
}
```

Assume that the attacker wants to execute malicious instructions at address `0xDEADBEEF`. The attacker should write a carefully-chosen input that will override the value written into RIP, which is 24 bytes away from `name`.

One of the inputs could be:

```
'A' * 24 +
'\xEF\xBE\xAD\xDE'

# Note \xEF denotes EF in hexadecimal.l
```

Question: Is the x86 calling convention used everywhere or is it used only in this class? If not how do we know what the stack looks like and where rip is.? GDB

3.3 Putting Together an Attack

1. Find a memory safety vulnerability
2. write malicious shellcode at an known memory address
3. overwrite the RIP with the address of the shellcode
4. return from the function begin executing the written malicious shellcode

Definition 3.3: shellcode

Shellcode is a small piece of machine code typically written in assembly language that is designed to be injected and executed directly in a computer's memory. It's often used in the context of exploiting software vulnerabilities to gain unauthorized access to a computer system. Here are some key points about shellcode:

For the code:

```
1 void vulnerable(void) {
2     char name[20];
3     gets(name);
4 }
```

Here is what the stack could look like if we write the shellcode at the beginning of name, an the rest is garbage ('A').

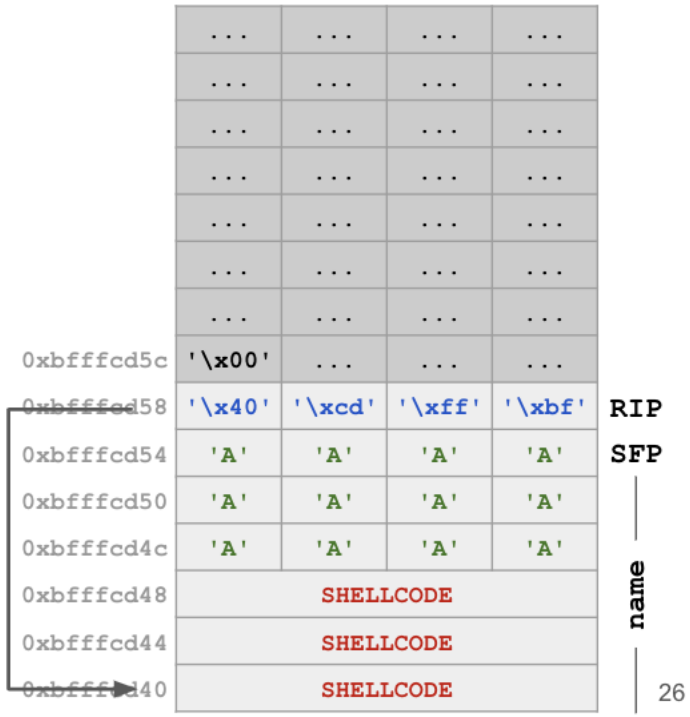


Figure 3.1: The stack with shell code written at the beginning of name.

Writing Safer Code

Realize that `gets` is not safe. A safer function that reads input from the user is `fgets` which requires the program to input exactly how many bites they could change.

In general, read up on the functions to see if they they have vulnerabilities in the `man pages`

Integer Memory Safety Vulnerabilities

This is an attacker where the attacker exploits how integers are represented in C memory.

Consider the program:

```c
void function (int len, char * data) {
    char buf[64];
    if (len > 64) return;
    memcpy(buff, data, len);
}
```

If we look at the function signature of memcpy:
`void *memcpy(void* dest, const void *src, size_t n);`

09/06/23

Lecture 4

More Memory Vulnerabilities

4.1 `printf` vulnerabilities

Consider the program:

```c
void func(void) {
    int secret = 42;
    printf("%d\n");
}
```

There is a mismatch between the format string specifiers and the arguments to printf. Printf expects a number to be in the second argument. So it will look 8 bytes above the rip of printf, to find the second argument (arguments are in reverse order). The stack will look like this:

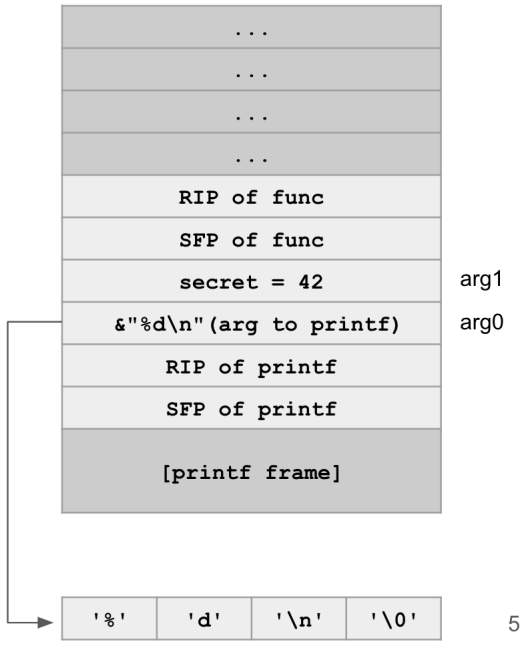


Figure 4.1: func(void) stack

You can also write values using the %n specifier:

%n treats the next argument as a **pointer** and writes the number of bytes printed so far to that address (usually used to calculate output spacing)

Example 4.1: %n example

Consider the program:

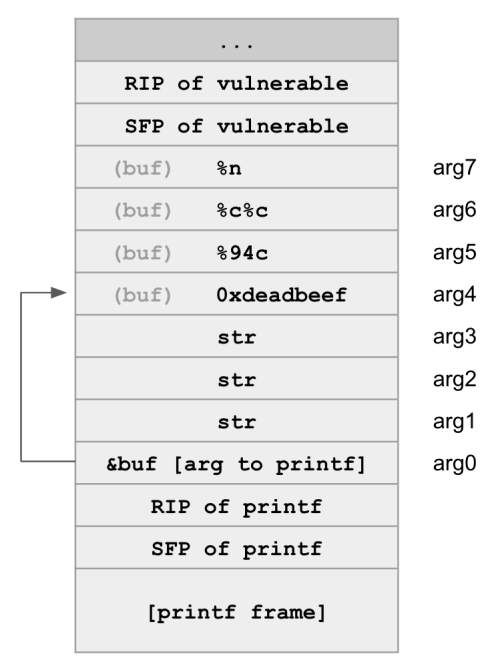
```
1 void vulnerable(void) {
2     char buf[16];
3     char str[12];
4     fgets(buf, 16, stdin);
5     printf(buf);
6 }
```

Question: Construct an input to write 100 to address 0xdeadbeef

Solution:

Input:	0xdeadbeef	%94c	%c	%c	%n
# chars used:	4	4	2	2	2
Consumes:	N/A	arg1	arg2	arg3	arg4
# bytes printed:	4	94	1	1	0

Stack diagram:



Note %94c prints the next argument on the stack as a character, padded to 94 bytes

4.1.1 Defense Against Printf Vulnerabilities

Do not give the user control to the first argument of print f. Here is safe code

```
void not_vulnerable(void) {
    chat buf[64]
    if (fgets(buf, 64, stdin) == NULL)
        return;
    printf("%s", buf);
}
```

4.2 Off-by-one Vulnerabilities

Consider the function:

```
void vulnerable(void) {
    char name[20];
    fread(name, 21, 1, stdin);
}
```

`name` holds 20 bytes, but we allow the attacker to write 21 bytes. Is it possible that an attacker could execute shellcode just by changing one byte?

The attacker is of course able to overwrite all 20 bytes of name, but also the least significant byte of the SFP of vulnerable. If the

09/11/23

Lecture 5

Mitigating Memory Safety Vulnerabilities

1. Using a memory-safe language. This is the only way to stop 100% of memory safety vulnerabilities.
2. Writing memory-safe code, like checking a ptr is not null before dereference. Using safe libraries.

5.1 Mitigation: Non-executable pages/ Write XOR Execute

Common buffer overflow exploits involve the attacker placing shellcode somewhere in the stack and overwriting the rip to cause the program to execute that code. One way to defend against this category of attacks is to make some portions of memory non-executable, which means that data in these regions should not be interpreted as CPU instructions.

The code section of memory is executable but not writable. The stack is writable but not executable.

5.2 Subverting Non-executable pages: Return into libc

Most C programs import libraries with thousands or even millions lines of instructions, all of which are marked as executable. An attacker can exploit this by overwriting the rip to point to a desired C library function, and some take arguments. The attacker can carefully place the desired arguments to the library function on the stack.

5.3 Subverting non-executable pages: Return-oriented programming

We can construct a custom shellcode using pieces of code, called gadgets, that already exist in memory. These gadgets are not functions, they don't need to start with a prologue or end with an epilogue, just as long as they end with a `ret` instruction. The general strategy for executing ROPs is to write a chain of return addresses at the RIP to

achieve the behavior that we want. Each time we jump to an ROP gadget, we eventually execute the `ret` instruction and then pop the next return address of the stack, jumping to the next gadget.

5.4 Mitigation: Stack canaries

a canary bird is a sacrificial animal commonly used by coal miners to alert when toxic gas builds up. We can use the same idea to prevent against buffer overflow attacks. The compiler places a canary value (stack canary) on the stack. This value is not used by the function so it should remain unchanged. After the function returns, if the cary value has changed, then it's like the canary bird died in the coal mine, so something must've gone wrong, so the program will crash before any further damage is done.

The stack canary is a random value generated at runtime. The canary is 1 world long. Stack canaries are usually guaranteed to contain a null byte (usually the first byte). This lets the canary defend against string-based memory safety exploits.

5.5 Subverting Stack Canaries

There are many exploits that the stack canary cannot detect:

- Can't defend against attacks outside of the stack, like vulnerable heap memory.
- Stack canaries don't stop an attacker from overwriting local variables. Recall the `authenticated` example.
- Some exploits do not write to non-consecutive parts of memory. For example, format string vulnerabilities let an attacker write directly to the rip without having to overwrite everything between a local variable and the rip, so the canary value is unchanged.

9/11/23

Lecture 6

Intro to Cryptography

Cryptography is the study of how to write secret messages. To formally study cryptography, we have to define a mathematically rigorous framework that lets us analyze the security of various cryptographic schemes.

Meet Alice, Bob, Eve, and Mallory. The most basic cryptography is one of ensuring the security of communications across an insecure medium. Alice and Bob wish to communicate thru a channel which insecure. The channel is subject to eavesdropping by Eve. In some settings, Eve may be replaced by an active adversary *Mallory*, who can tamper with communications in addition to eavesdropping. The goal is for Alice and Bob to communicate in such a way that Eve has no clue about the contents of their exchange, and Mallory is unable to tamper with the contents of their exchange without being detected.

Definition 6.1: Key

The most basic building block of any cryptographic system is the key. It's a secret value that helps us secure messages. There are two main key models in modern cryptography:

1. The *symmetric key model*, Alice and Bob both know the value of a secret key, and must secure their communications using this shared secret value.
2. The *asymmetric key model*, each person has a secret key and a corresponding public key, recall RSA.

In cryptography there are three main security properties/goals we want to achieve

1. **Confidentiality**: The property that prevents adversaries from **reading** our private data.
2. **Integrity**: The property that prevents adversaries from **tampering** with our private data without being detected.

3. **Authenticity**: The property that lets us determine who created a given message.

6.0.1 Integrity (and Authenticity)

Schemes provide integrity by adding a **tag** or **signature** on messages. Bob receives the message from Alice and checks if the tag/signature is valid. If it's not, then the authenticity of the message received is not guaranteed.

Definition 6.2: Kerckhoff's Security Principle

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

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

Here is the model we'll be working with:

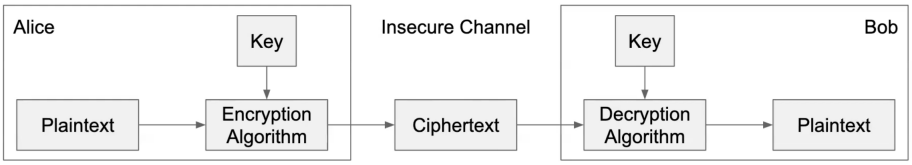


Figure 6.1

6.1 Threat Models

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

- **ciphertext-only attack**: Eve has intercepted a single encrypted message and wishes to recover the **plaintext** (the original message).
- **chosen-plaintext attack**: Eve can trick Alice to encrypt arbitrary messages of Eve's choice, for which Eve can then observe the resulting ciphertext. (This might happen if Eve has access to the encryption system, or can generate external events that will lead Alice to sending predictable messages in response.) At some other point in time, Alice encrypts a message that is unknown to Eve; Eve intercepts the encryption of Alice's message and aims to recover the message given what Eve has observed about previous encryptions.

9/19/23

Lecture 7

Block Ciphers and Modes of Operation

7.1 Symmetric-Key Encryption

A symmetric-key encryption scheme has three algorithms:

- KeyGen() $\to K$: Generate a key K
- Enc$(K, M) \to C$: Encrypt a **plaintext** M using the key K to produce **ciphertext** C.
- Dec$(K, C) \to M$: Decrypt a ciphertext C using the key K.

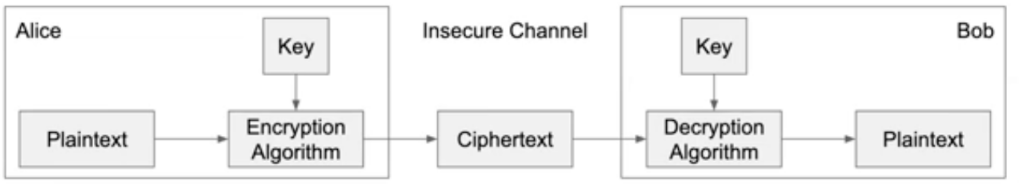


Figure 7.1: Symmetric Key Model

7.1.1 Properties of Symmetric Encryption Scheme

- **correctness:** Decrypting a ciphertext should result in the message that was originally encrypted.
  - $(\forall K, M)\ Dec(K, Enc(K, M)) = M$
- Efficiency: Enc, Dec algorithms should be fast
- Security: confidentiality

Definition 7.1: Confidentiality

A more rigorous definition is as follows: the ciphertext C should not give the attacker *any additional information* about the plaintext message M.

We can further formalize this definition by designing an experiment to test whether Eve has learned any additional information: if Eve can guess the ciphertext to be either any of n messages, when Eve sees the ciphertext, the probability that Eve chooses the correct message is $1/n$, which is no different than if Eve hadn't seen the ciphertext at all.

If we achieve confidentiality, meaning that; even though Eve can trick Alice into encrypting some messages, she still cannot distinguish whether Alice sent message M_0 or message M_1 in the experiment. This definition is known as **Indistinguishably under chosen plaintext attack (IND-CPA)**. We can use an experiment or game, played between the adversary Eve and the challenger Alice, to formally prove that a given encryption scheme is IND-CPA secure, or show that it is not IND-CPA secure.

7.2 XOR Review

Symmetric-key encryption often relies on the bitwise XOR operator (written as $\oplus$). Here are some useful properties:

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

7.3 One Time Pad; *a symmetric encryption*

The OTP scheme is a simple and idealized encryption scheme that helps illustrate some important concepts, though it is impractical for real-world use, as we'll see shortly. In the one-time pad scheme, Alice and Bob share an n-bit secret key $K = k_1 \dots k_n$, generated every time Alice wants to encrypt a message, where each bit is picked uniformly at random.

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

The desired properties of the encryption scheme are:

1. It should scramble up the message, i.e., map it a ciphertext $C = c_1 \dots c_n$
2. Given knowledge of the secret key K, it should be easy to recover M from C.

3. Eve, who does not know K, should get *no* information about M

The algorithms for OTP are as follows:

- $KeyGen()$, randomly generate an n-bit key.
- $Enc(K, M) = K \oplus M \rightarrow$, the jth bit of the ciphertext is the jth bit of the message, XOR with the jth bit of the key.
- $Dec(K, C) = K \oplus C$

7.3.1 How Secure is OTP?

If we were to reuse the same key for every encryption, we can show that this scheme is not secure: Eve would receive a ciphertext C and trick Eve into encrypting M_0, if C matches M_0, then Eve would guess M_0 is the message, otherwise, she guesses M_1. In summary, eve guesses the correct message with 100% probability.

If we were to generate a random keys every time, we can show that this scheme is perfectly secure. However, random get generation is often expensive, and more over, when we generate the random key, we have to share this key with Bob in a secure way anyways, so why not send the message with this secure scheme.

7.4 Block Ciphers

Generating new keys for every encryption is difficult and expensive. Instead, Alice and Bob in most symmetric encryption schemes use a single key to repeatedly encrypt and decrypt messages. The block cipher is a fundamental building block in implementing such a symmetric scheme.

Definition 7.2: Block Cipher

An encryption/decryption algorithm that encrypts a fixed-size block of bits.
Properties:

- **Correctness**: E_k is a permutation (bijective function) on n-bit strings, D_k is its inverse
- **Efficiency**: Encryption/decryption should be fast
- **Security**: The encryption function E behaves like a random permutation.

7.4.1 Security

The block ciphers are not IND-CPA secure, because it's deterministic. No deterministic scheme can be IND-CPA secure because the adversary can always tell if the same message was encrypted twice. Moreover, another issue with block ciphers can only encrypt messages of a fixed size, usually 128-bit messages. We'll address these issues with **modes of operations**.

7.5 Block Cipher Modes of Operation

In order to support encrpytion of longer messages, we can split the message into multi 128 bits, and apply a block sipher to each substring, and then concatinating the ciphertext to obtain encrypted code.

9/20/23

Lecture 8

Cryptographic Hashes and MACs

8.1 Cryptographic Hashes

Definition 8.1: Hash Function

A cryptographic hash function is a function, H, that when applied on a message, M, can be used to generate a fixed-length "fingerprint" of the message. The hash function, H, is deterministic, meaning if you compute $H(M)$ twice with the same input M, you will always get the same hash value twice. The hash function is unkeyed.

For hash functions, since the produce "finger prints", any change to the message, no matter how small, will change many of the bits of the hash value with there being no detectable patterns as to how the output changes based on specific input changes.

8.1.1 Properties of Hash Functions

Some of the most significant properties include the following

- **One-way**: Given x, it is easy to compute $H(x)$. However, given a hash value y, it is unfeasible to find any input x such that $H(x) = y$. This is also known as $pre = imageresistant$.
- **Second Preimage Resistant**: Given x, it is infeasible to find another input x' such that $x \neq x'$ but $H(x) = H(x')$
- **Collision Resistant**: It's infeasible to find *any pair of messages* x, x' such that $x \neq x'$ but $H(x) = H(x')$. This is very similar to the second property but now the adversary can freely choose their starting point x.

Note: "infeasible" means that there is no known way to accomplish it with any realistic amount of computing power. Also the third property implies the second.

8.1.2 Hash Algorithms

Today there are two primary "families" of hash algorithms in common use that are believed to be secure: SHA2 and SHA3. The only significant different is that SHA2 is vulnerable to a *length extension attack*.

8.1.3 Do Hashes provide integrity

It depends on your threat model. If Alice wants to communicate with Bob over an insecure channel where Mallory is capable or tampering with the message, hashes do not provide integrity. Mallory can simply change the message and the hash value so that when bob checks the hash of the tampered message recieved and the hash of the tampered message, it will be the same even though the message is tampered with.

This is because there is no key, no secrecy to the hash function. Anyone can compute the hash of a message. But can we use hash functions as a building blocks? Yes in MACs,

8.2 Message Authentication Codes

MACS guarantee integrity and authenticity to enable the recipient of a message to detect spoofing and tampering.

Definition 8.2

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

checksum invalid. Formally, the MAC, T, on a message M, is a value $F(K, M) = T$, called the tag for M or the MAC of M.

When Alice wants to send a message with integrity and authentication, she first computes a MAC on the message $T_{ag} = \text{MAC}(K, M)$. She sends the message and the mac $\langle M, T_{ag} \rangle$. When bob recieves the message, he will recompute $\text{MAC}(K, M)$ and check that it makes the tag he received. If the tags don't match, he will ignore the message and presume that some tampering or message corruption occurred.

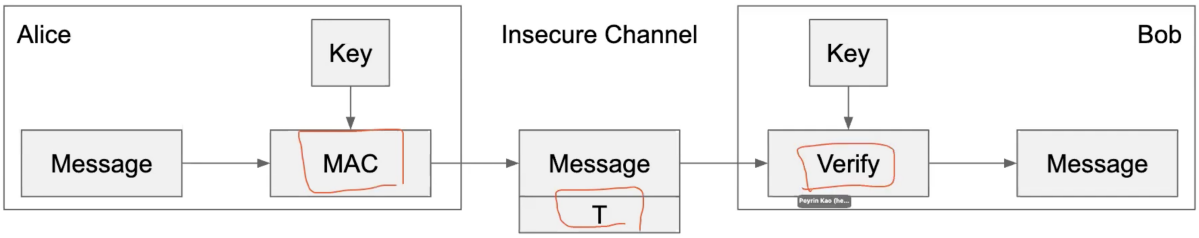


Figure 8.1

8.3 AES-EMAC

How do we build secure MACs? There are multiple schemes out there but a good on is AES-CMAC, an algorithm standardized by NIST. But lets look at AES-EMAC; it is a slightly simplified version of AES-CMAC that retains its essential character but differs in a few details.

In AES-EMAC, the key K is 256 bits, viewed as a pair of 128-bit AES keys: $K = \langle K_1, K_2 \rangle$.

What are AES keys?

The message M is decomposed into a sequence of 128-bit blocks: