Introduction

Confidentiality - you don't leak any confidential information
Integrity - information remains untouched
Availability - system has to remain available even if attacker tries to stop it

You cannot tell if system is secure or not without the threat model

ILOVEYOU script security problems:

- curiosity is easily abused
- no warning that attachment can be executed
- program from untrusted source can do sensitive operations
  threat model: script kiddies - everyone could write this

Heartbleed:

- short message ("HAT"), but is requesting long message ("500 characters")
- this is why it sends information next to the message. That next message was part of logs, which stored keys and other sensitive information.
- also it cannot be known if this happens, thus it cannot be known if and how many times it was done
  security problem: C is unsafe
- relies on programmer to check bounds
- programmers make mistakes
  impact: loss of confidentiality
  threat model: skilled hacker

DDos attack security (through botnet) - many embedded devices have default passwords, and it is hard to distinguish legitimate and malicious requests
impact: loss of availability
threat model - script kiddie - once you have botnet, attack is trivial; botnet nodes can be rented on black market

stuxnet - malware

- abused several previously unknown windows bugs (zero day - too late to fix)
- spread over the internet and through usb devices as well (that is how it reached the airgaped devices)
- spreads to industrial devices
- hides itself from OS (rootkit)
  it was harmless on most of the computers
  purpose: targeted nuclear program in Iran (spread to countries starting with I), and specifically uranium enrichment devices
  it would not stand out and in time would damage these devices
  because of the knowledge and sophistication it implies US and/or Israeli secret service and/or Dutch
  example or cyberwarfare
  security problems:
- bugs and insecure configs in Windows and other software
- giving control over certificate signing key
- hard to install updates on industrial conrol systems
  impact: loss of availability
  threat model: advanced persistent threat

Threat model:
defines what an attacker can and cannot do
e.g.: technical ability? log in to system? intercept connections? physically control the hardware? insert backdoors ahead of time? (- supply chain attack - backdoor inserted during supply chain)

A system is secure if CIA properties cannot be violeted within the threat model
Well defined threat model helps to define weaknesses and develop a plan how to fix them

Why is security hard?:
Asymmetry: hacker need to find one weakness but developer needs to find all weaknesses
Hard to convince managers: since it can increase cost, can decrease user friendliness and hard to measure, since it could be invisible until attacked
Many levels: hardware, OS, applications, frameworks, ...

Lecture 3

Revision

Example

```
int main(int argc, char **argv){
        char buf[64];
        int len = atoi(argv[1]);
        read(STDIN_FILENO, buf, len);
        write(STDOUT_FILENO, buf, len);
}
```

- **Fault** - No check for the size of the input in bounds
- **Error** - If the length larger than buffer, it can overwrite the stack behind the buffer
- **Vulnerability** - buffer overflow; missing length check
- **Exploit** - if the length is > 64 (this is because read/write works with binary data, so no null byte), it can go over the buffer
- **Impact** - integrity, availability; confidentiality - depends on the context, ie if there is a chance for privelage escalation. If there is a chance - there is a chance for arbitrary code execution
- how to **fix** it - add a missing check.

Types of memory

- parameter - in stack or a register (deallocation when the function returns). Since it is a function parameter, multiple of these can happen when 1) multithreaded part, 2) it is recursive.
- size_t variable - in stack -||-
- (char *) varialbe - in stack, but points to the heap. There can be multiple of these just if the function is called multiple times and not freed, since it is on the heap.
- dereferenced (char *) - in the heap. Deallocated when free is called. Is it a bug if it is not freed in the same function? - Depends on the context,

since heap vars can be deallocated later.

Programming languages

Choosing the right programming language is important first step

Programming languages are ofthen classified as:

- unsafe languages trust programmer to know what they are doing (e.g. assembly, C, C++, older languages)
- safe languages reduce the impact of programmer mistakes (e.g. C#, Java, Javascript, PHP, Python, Rust, etc.)

	Unsafe	**Safe**
Deallocation	manual	garbage collected
Integer overflow	wraps around	exception
Out-of-bound access	undefined	exception
Pointers	allowed	only references
Unsafe typecasts	allowed	disallowed or automatic conversion
Variable initialization	optional	enforced or automatic

Usually you should use safe languages, but other uses that are these are not suitable for:

- performance-critical code
- low-level code (direct interaction with OS or hardware) - (it is becoming more possible with Rust)
- Resource-constrained embedded systems
- Real-time systems (performance less predictable with safe languages. E.g. with older Java there could be a situation when garbage collection would take few seconds)
- There is already a lot of legacy code

Sometimes it is possible to combine both. E.g. NumPy library that is written in C for Python. Then it is just important that library interfaces are safe.

C++ is unsafe but has features associated with safe langs

C# is safe but allows specifically marked unsafe code

Safe languages can be interpreted (Python/PHP), just-in-time compiled (Java/C#), or native compiled (Rust)

Formal verification: idea is to have a formal proof of security. This requires complete and correct specification; program written in a suitable language (typically functional), and lots of time. It is very impractical.

It provides strong guarantees but because of its requirements, not widely used. Most suitable when:

- system is very simple
- system is mass-produced on large scale to spread cost
- security is critical, esp if a matter of life and death (pacemaker, insulin pumps)
- hardware critical

Secure software design

Many design decision related to security should be taken in to account before writing the first line of code.

Number of bugs generally is proportional to complexity. Even in mature software. Thus, less code means more security.

Add only features that are certain to be necessary. 1) Fewer features means less opportunity for erros. 2) Rarely used features will be poorly tested.

Also this is known as KISS (Keep It Simple, Stupid)

Program is most vulnerable where untrusted data is processed.
This is known as "attack surface". Program defense is more effective with small attack surface.

How to reduce the surface:

- few interfaces, as simple as possible
- early validation of untrusted data
- keep messages similar adn use same code to handle all cases (Do not copy paste code)
- Restrict interfaces to authorized users.

Compartmentalization:

Applications often naturally consist of compartments (client, server, db). Usually attacker aims for the most data-ful compartment, but all compartments should be secured.

Processes cannot access each other's memory

Vulnerabilities only cross process throygh explicit interfaces (networking, files, inter-process communication)

Principle of least privilege (POLP)\

Every program and every privileged user of the system should operate using the least amount of privilege necessary to complete the job\

If a web-server is compromised. Solution is to drop privileges.

Databases - if a program can execute arbitrary SQL queries on the db, it does not serve as a separate component.

Solution: separate database user dor each component accessing db; Assign permissions per table (or even column) and user

Better: forbid arbitrary SQL queries entirely; create stored procedures and control access to them

Android security: while on Windows/Linux programs are allowed to do everything the user that started then can do; On Androind, each app only get specifically assigned permissions

Make sure the defaults are as secure as possible.

E.g. ubuntu forbids logging in as root user by default, creating only a user account with sudo access.

Entropy

4 letters, one of which uppercase + 4 numbers: log2(26^4 * 4 * 10^4) possibilities
3 words in 8192 word dictionary: ($2^{13)}$3 possibilities

Cryptography

Cryptography goals

1. Confidentiality - Eve cannot read Alice's message
2. Integrity - Mallory cannot alter Alice's message without Bob finding out
3. Authentification - Mallory cannot convince Bob she is Alice
4. Non-repudiation - Mallory cannot deny she was the one who sent a message to Bob

Terms:\

- Plaintext - the readable text to be transmitted
- ciphertext - the unreadable text actually sent
- encryption - plain -> cypher
- decryption - cypher -> plain

Kerckhoffs' principle: when doing encryption, separate the algorithm from the key. Use public key.

Attacks

Brute Force:
attempt to decryot the message with every possible key.
Solution: Larger Key Space (e.g. instead of shifting alphabet, we create a random

mapping).

Frequency Analysis:
find the most common letter (probably e), find the second most common letter (probably a).... Eventually find plaintext and a key.

Cyptanalysis:

- known ciphertext is used to determine the key
- known plaintext is used to determine the key
- chosen plaintext is used to determine the key (strongest attacker)

Modern Encryption:
Cyphers are insecure these days. they can be used for obfuscation.
Now block encodings are used widely.

Confidentiality:
Ideal approach is *end-to-end encryption*. This is such that only sender and the receiver can have the key and the plaintext.

Integrity:
Solution is *digital signatures*. Sequence of bits that are linked to the message and the sender. Only the sender is able to produce the same sequence of bits.

Authentication:
Both confidentiality and integrity is useless if this is not done. We need to know who produced the data
Man-in-the-middle attack is if connection can be intercepted. If Mallory intercepts login and uses the same password and communicates with the

Non-repudaition:
Solution - *asymmetric digital signatures*. There are different keys to sign and verify. If these are symmetric keys, the other end can sign it, so Bob cannot prove that Alice signed it to someone else, because he can sign it himself, not only verify it.

Cryptographic hashes

Hash function maps the message to hash. This has to be a one-way function, so

there is no way to get back the original message. That is why it is hard to reverse. Hash is a secure way to verify a message.

Properties

- pre-image resistance - it is hard to find message m such that hash (m) = h
- srcond pre-image resistance - given a message m1 is ti s hard to find another message m2 such that hash m1 = hash m2. Prevents attacker from creating a message with same signature as signed message
- collision resistance - it is hard to find two different messages m1 and m2 such that hash m1 = hash m2.

Well known hashes: SHA-2, SHA-3, MD5

Random numbers

Many crypto algorithms need random numbers.
Problem is that CPU is deterministic and it is hard to produce random data. So it should use information from the environment.\

That is why pseudorandom number generator is used. It uses a seed which is updated with each new number. \

However sequence is often predictable. that is why it is necessary to use cryptographically secure PRNG\

Symmetric cryptography

Alice and Bob know the key, Mallory does not.

I missed some parts

Stream cipher:
One time pad rarely practival - huge key size and no key reuse.

Solution: use a PRNG to generate key. Initial seed now becomes the real key, and it is simple and efficient.

Block ciphers
Idea: divide data in blocks, perform computation using key to map each plain block into a cipher block, reverse computation possible with the same key.
It is widely used.

Cipher Block Chaining (CBC):
Block-by-block encryption is Electronic Codebook. It reveals repetitions, and allows reordering blocks.
Solution - make each block edepend on the previous one. For the first block use initialization vector (IV) instead (this should not be reused, doesn't have to be a secret).

How it works: \

- it xors IV with plaintext, and apply the key. That is how we get ciphertext.
- then use the ciphertext to xor the next plaintext. etc etc

There is a need of padding. Can be fillers with numbers (07 07 07 ... 07)

Padding Oracle Attack:
Assumptions:

1. Attacker can send encrypted message to server
2. Server decrypts message, returning that there is an error message in the last block or not, depending on the padding.
3. **Missed it**
   This means that we can decrypt the last block. Then we send the message one block shorter, and we decrypt 2nd to last block. This will make it that we decrypt almost the whole message.

Solution: make an error indistinguishable to other data. Such that attacker would not know that they can exploit it.

Symmetric signatures

Message authentication code (MAC) is signature for message. - If message is altered, MAC is no longer valid. Key is needed to generate valid MAC, and same key is needed to validate MAC.
Typical example: Hash MAC
Signing: Alice combines key with message, then applies hash function. Verification: Bob .....

Asymmetric Cryptography

Asymmetric crypto uses two keys, this is used to verify with a public key who signed it

RSA

Bob creates his own key - public + private keys. Public key does not need to be hidde, but private key has to be hidden.

Encryption function is designed in such a way that Epr(Epu(m))=m. Key property - pr is not derivable from pu.
RSA is based on the modular arithmetic. It is exponentiation and then mod.
Due to complex math, computing pr from pu with message requeires writing message as a product of prime numbers.

1. Alice determines Bob's public key e. This needs to be received from Bob personally or received from trusted third party who verified Bob's identity. This is critical from man-in-the-middle attack.
2. Alice enrypts message with e.
3. Sends message
4. Bob decrypts with his private key.

Messages require padding, because if message is short it is not mathematically difficult to decrypt. Padding is random padding, but this is handled by the

algortihm itself.

Important notes:\

- random padding is critical so the ciphertext is different each time
- RSA only works for data that fits inside the key size, not multiple blocks

Typical use:\

- generate symmetric key
- RSA encrypts only symmetric key
- actual data encrypted with symmetric key

RSA digital signatures.
Signing: Alice computes hash(m), encrypts it with her private key d,
Alice sends m and Ed(hash(m)) to Bob.
Verification: Bob determines Alice's pub key, ...

Integrity protected:\

- if Mallory modifies m, the hash is no longer correct
- .....

RSA depends on inability to factor large primes efficiently. Problem arises when there is **Quantum-Computing**.

Post-Quantum Crypto - for now it is relatively small field, but it is a big reasearch topic.

Quantum Crypto:
future computers may communicate in ways that are physically impossible to intercept. It would not rely on maths but on physics.

Key management

We have to be confident that the key is correct person's key.

Ways to solve this:

- have a separate trusted channel (usb key - Bob gives it to Alice)
- use trusted third party to authenticate keys

Key Distribution Centre - KDC
Have a trusted party T:\

- It shares symmetric keys with all others
- verifies identities of key owners

E.g.: Alice wants to Bob, so it gets the key from KDC.

Kat/Kbt - key shared between Alice with T/Bob with T
Kab - temporary key between Alice and Bob
N - random number that is not reused (nonce) \

Communication is encrypted with Kab.

Process of setup:\

1. A -> T, to send message to Bob (A,B)
2. T -> A, encrypted message that contains Kab, and encrypted message (Alice, Kab), encrypted with Bob's shared key.
3. A -> B, encypted message with Bob's shared key, that they want to talk
4. B -> A, N encypted with Kab
5. A -> B, N-1 encrypted with Kab (to prove that Alice really has Kab)

This is not an existing protocol, just simplification. In reality it is needed to have more nonces and shared keys, so that replay attack would be prevented. Needham-Shroeder protocol. And on top of this Kerberos protocol is built which is widely used in Windows and Unix.

Kerberos drawbacks:\

- needs constant access to KDC
- Single point of failure: availability or all keys can be compromised by an attack
- strict timeouts due to replay attacks: which means that clocks must be in sync, and need to get new key frequently.

Public Key Infrastructure - PKI
KDC used symmetric crypto, but this could be asymmetric crypto.

Certification Authoritites: - issue certificates and keeps *revocation list* of certificates that may be compromised. (IT DOES NOT TOUCH PRIVATE KEYS - that is why it is a bit safer).
We need to trust these companies if we want to use this. We do this usually because OSes choose it.

Compromised CA allow forged certificates.

X5.09 Certificates
these are used to prove identity was verified.
They contain\

- identity of holder
- domain name of holder
- pub key of holder
- expiration date
- signature from CA

Getting a Certificate:\

1. Bob creates key pair
2. Bob creates a certificate signing request including data to be signed (especially the pub key)
3. CA verifies Bob's identity
4. CA creates certificate using CSR and signs with its private key
5. Bob can now use certificate to prove public key is his

Self-Signed Certificates:
it is signed with its own private key.

Can be made by anyone, since it saves money but is not trusted by default (doesn't mean it is insecure).
Programs can be configured to trust specific certificates. It is usefull if pub key can be verified personally.
Example of this could be in SSL.

Checking certificate:\

1. server address must match the certificate
2. srrvers must demonstrate knowledge of private key matching the pub key (e.g decrypt data encrypted with pub key or create signature valid with pub key)
3. certificate must not be expired
4. signature from CA must be valid
5. CA must be trusted byt the user
6. check if it is not revoked

Symmetric encryption key establishment
With symmetric encryption, we must first agree on a key.
It is possible to find a shared key over the network without sending any info from which that key can be derived\

Diffie-Hellman Key Exchange: this is a setup of a two keys (from both sides) that cannot be decrypted by anyone else. This is because it is hard to undo exponentiation.
Key part of this is that Alice has key g^a mod m, Bob has g^b mod m, then they exchange these, and both now have a key g^(a*b) mod m.
It still cannot be used alone, because this is still vulnerable to man-in-the-middle attack.

We need ecryption on the Web.
There can be anyone eavesdropping on HTTP. In ISPs, hackers, governments, etc.

Secure Socket Layer (SSL) ensures cryptographic protection for a network connection on top of TCP.
Any TCP protocol can be modified to support SSL. That is why we use HTTPS.

SSL goals:\

- end-to-end encryption that ensures only browser and server can read.
- typically authenticates the server using X5.09 certificate.
- Typically does not authenticate the client. Since the server can to this in the application layer. (when the keys are setup we already know that we are talking with the server).

Passwords

MY LAPTOP DIED, GO THROUGH THE SLIDES HERE

How to check certificate validity?:\

- server address must match the certificate
- server must demostrate knowledge of private key matching public key
- Certificates must not be expired
- signature from CA must be valid
- CA must be trusted by the user
- certificate must not be on revocation list

Conservative Programming

Trying to prevent vulnerabilities.\

Handling user input

URL is a user input, thus has to be careful - never trust the parameters from user input blindly and verify the user authority for such input.
This is risky when handling direct user input, but after storing a message in db, and retrieving it, it still is risky.
Solution - sanitization (if something that shouldn't be there, drop/remove), escaping (e.g. SQL input), authorization (only certain user can access such parameters).\

Sanitization: it is for the things that are illegitimate\

- verify that data follows the expected structure
- verify that any values are within reasonable range

Be careful with error messages, such that attacker shouldn't have anything to exploit from information.\

Name should not probably be of length 1000, or empty strings can cause troubles as well

Name should probably not contain < >\

Verify numbers to be in a reasonable range\

Some input may only take values from a specific set\

USE REGEXES for allowlisting\

Escaping\

characters that have special meanings, this is replaced such that they lose special meaning\

Intrusion detection

Validation might not be sufficient to detect intrusion, because e.g. there can be spam emails.\

The heuristics depend on the context of the system.\

Skipped a lot in the lecture, internet was shit

Questions

1000000 begning files
5000 malicious files
sys A: 5% false positives, 10% false negatives. (false positive - non malicious

marked as malicious, false negative - malicious not marked as malicious)
sys B: 4% false positives, 50% false negatives.\

Which is more safe?
System B.\

Why choose another system?
Depends on the context - in some cases false positives are important, but if system is vulnerable false negatives could be quite bad.\

What should the server verify before trusting data sent by the client for:\

- data to be stored/processed by the server - validation/escaping in the client, but client should be considered untrusted, so it should be on the server.
- secrets between the client and the server - use trusted third party to verify the secret to prevent the man-in-the-middle attack. Also prevent others to forge the secret.
- Requested page address - verify if that user is allowed to access that page (knowing the url is not sufficient)

Checking assumptions

Smart car key example

If key is close to the car, car automatically unlocks. So if the key is far away but the signal is amplified, the car still can be unlocked. Assumption is that the key, thus the owner is close.\

There are a lot of programming assumptions. Such as: person's age fits in three characters, string holds the path to a file we are allowed to read, the object has not been freed yet (this is common for use-after-free vulnerabilities)\

Use assert for the assumption that you make\

If the assumption is violated, there is no need to keep the program running,

because the program is no longer safe.\

It is also important to document the assumptions, and also check it. It also helps to debug the program.\

The assertions can be disabled with -DNDEBUG flag to increase performance, and it is usually done during deployment, but for security purposes it is better to keep them in.\

Don't make assumptions on:\

- User input
- data read from disk
- data received from the network

Order of Operations is a common source of bad assumptions. This is why threading might cause issues:
This is because user may give commands in unexpected sequence; Environment may change right after check; thread interleaving can be unexpected; signal may get delivered at bad time (just before blocking call);\

To avoid this - always verify whether system is in expected state before executing command, and fail if it is not.
Any interaction with the file system, network, etc, might see different state than the orevious one.\

Handling Errors

There are a lot of way for program to have errors. Almost all calls can fail, usually in multiple ways.\

System calls is a part of the system, that is out of hour control, therefore it is generally expected that anything can fail.\

Error handling goals

Even if an error happens we should ensure:\

- Any resources (memory, open files, mutexes, etc..) are still released.
- Persistent state (files/databases) remains consistent
- Program continues to work (if possible)
- Program remains secure (does not give a chance to bypass authentication)
- Program state remains consistent

Usually is it also desirable to inform the user and/or log the error for later analysis or debugging. But it might cause some vulnerabilities.\

How not to do error handling

Terminating immediately leaves no way to clean up resources and persistent state \

```
$handle = fopen('myfile.txt', 'r') or die('open fauled');
```

Ignoring is easy in C but means state will be inconsistent with expectations, which may threaten security\

```
FILE *file = fopen("myfile.txt", "r");
fread(buf, sizeof(buf), 1, file); // buf uninitialized
```

Bash is one of the languages that ignores errors and continues execution, which might lead to some big problems\

Functions report errors by: return value (C, PHP, UNIX API, Windows API); Excpetion (C++, C#, Java, Javascript, Python); Error code in memory (many third-party libraries)\

Handling errors is hard:

- always check for errors

- number of error handling paths grows quickly with number of calls that may fail
- each path produces a different path which might complicate cleanup

Example of unnoticed problem:\

```
*pwd_p = (char *)sqlite3_column_text(stmt, 0);
```

This points to some memmory in sqlite, ie statement object owns memory buffer storing string. This might get deallocated before use (use after free problem).\

To solve this we should make a copyt of the string:\

```
*pwd_p = strdup((char *)sqlite3_column_text(stmt, 0));
```

State consistency
Initializing reference parameters is one example of more general problem: consistency of state.
State is inconsistent when values are not what code expects them to be; Example - count inconsistent with number of entries in the list;

Before returning error code:\

- Think of all changes you made that affect program's state
- Reverse those changes to keep the state.

Exceptions

handling error codes gets messy quickly.

Exceptions provide a better alternative in most languages\

Exception handling\

Whenever an exception is thrown, execution continues at the exception handler of the most recent open try block\

- try block may be in another function

- code between exception throw and exception handler might not be executed

Cleanup is easier when handling exceptions because of catch blocks, or finally calls. It is nicer to have RAII practice in C++ (return call calls the destructor of an object)\

Exceptions can alter control flow on any call\

Memory management

Low-level languages allow and mostly require programmer to manage memory.

Types of memory:

- global
- stack
- heap (explicitly allocated - hard to manage)

Struct size can have padding to have alignement with the memory page size, thus one cannot rely on the direct calculation when doing malloc

Main issues is to allocate the right amount of heap memory (independent of the system)
Advice could be to use stack over heap if possible
In c++ advantages is new instead of malloc:

- new can fail and it would return NULL ptr
- new calls a contructor

Always use sizeof to compute size for malloc
When storing string in array, allocate extra element for terminator
Check whether malloc succeeds; it returns NULL in out-of-memory conditions, which attacker may be able to trigger

strcpy is bad, use strncpy
strncpy can lose a null terminator, so use snprintf

the problem that arises is that we might lose data after n bytes, so use strdup and free that temporary buffer later.

avoid functions that do not check buffer boundaries

verify allocation size is consisten with actual content

both reading and writing out-of-bounds unsafe (it might leak information that make attacks easier or might read code pointer and divert the execution)

Memory initialization

- global memory is automatically initialized
- stack memory not automatically initialized
- heap memory not initialized by malloc, but calloc does that

low-level languages do not automatically initialize memory (reading memory before writing to it can be abused by attackers to leak info or basis to corrupt memory)

advice: initialize everything at allocation time

C++ automiatically calls object constructors
Default constructor automatically initialize some fields but not all

In C++ it is important which class inherits which. (e.g. ClassB and ClassC inherits ClassA. ClassB is passed as ClassA obj, ClassC static casts as ClassA, but inside it is ClassB)

Memory access is type-unsafe if the value is read as another type than it was written
This usually happens due to incorrect typecasts.
In C there are no checking for typecasts. In C++ there is a dynamic*cast (it is slow) which ensures type safety.*
_*Advice* don't use casts if possible, and prefer dynamic_casts

Object lifetime

Incorrectly managed object lifetime results in memory leaks or use-after-free errors

Advice:

- use stack when possible
- never return pointer to static variable and never store it in data structures that survieve the functions
- in c++ use RAII
- c++ offers smart pointers that help manage heap memory
- after free/delete, set deallocated pointers to NULL

Smart pointers - are classes that look like pointers.
std::unique_ptr - prevents copying the pointer; frees object when pointer lifetime ends; full protection against UAF and memory leaks, but not always possible. Does not help too much but prevents vulnerable code
std::shared_ptr - everytime it is copied, it keeps track how many copies there are. When the smart ptr goes after scope, it decreases the count, and when count is 0, it is deallocated. Memory leaks can be possible (e.g. circular referencing)
std::weak_ptr - similar to shared_ptr but can go to NULL when shared_ptr count goes to 0

Integer overflows

when integers exceed range of type they are stored in result is incorrect - usually it wraps around.

It is an important type of attacks and can be used to bypass checks or overflow the buffer.

Handling numbers:

- validate numbers that come in, keep only the sensible amount
- choose types based on these ranges
- verify whether each input range before converting/storing it

Checking for overflow in arithmetic is hard in standard C/C++, because result of overflow is undefined.
Safe/efficient overflow checks as common extension (e.g. __builtin_add_overflow)

Secure Programming Development Tools

Compiler - detect and try to prevent possible vulnerabilites
Source control - prevent bugs due to group work and ensure bugs stay fixed

there are many testing tools as well

Compiler

Does static analysis.

It tracks control flow and fata flow through the program without actually executing it. (sometimes it is hard to do that esp with pointers)
It tries to understand what is happening but does not try to give false positives - it is conservative in reporting errors.

Advice:

- use -Wall and -Werror
- compiler warns about possible bugs found with static analysis
- e.g. can find uninitialized vars

Clang has more static analysis

Dangling pointer that is not detected by the compiler:

```
int *foo(void) {
        int a, *b = &a;
        return b;
}
```

Secure Programming LINT - splint

but even with this, it can miss, for example, out-of-bounds error

Dynamic Analysis

attempts to find buges at runtime - looks for undefined behaviour and kills the

program

this is inconvenient since you need explicit test cases

Testing

Designing tests

TEsting is critical for both correctness and security

Testing is mainly to cover the edge cases or the implicit/explicit assumptions.

Regression tests ensure program changes do not introduce new bugs - same test reused after changes such that checks if old bugs were not reintroduced

Usually tests would be derived from requirements.

Unit testing - many error/boundary conditions are hard to reach with just regular inputs (this is for testing specific parts, and this is because input is parsed/ sanitized and it is different)

To design effective tests, look further than expected inputs.

It is good to have meaningful errors.

Generate random inputs (following a pattern) to increase diversity in the ways the code is pushed to the limit.
It is step towards *fuzz testing*

Coverage

the goal is always to find all the bugs.

To solve that - find test cases that execute as much program code as possible.

How to get full line coverage? - try to get every line to execute with minimal amount of tests. So single if check will always be executed, but if-else might need

different checks.

How to get full statement coverage? - it is kind of the same as line coverage but deeper. To get amount how many tests you need is number of return statements.

Decision coverage - look for different conditional branches (if, else, while) and cover both true and false cases.

How to get full Decision coverage? - look for all cases and for each case come up with a test, and look if that case suits the other cases.
There can be optimizations in the compiler or expands the if statements. This is important for checks as well.
Example in code:

```
if(*s >= '0' && *s <= '9'){/*Do inside if */}
else
```

becomes

```
if(*s < '0') goto else_stmt;
if(*s > '9') goto else_stmt;
/*Do inside if */
```

Condition coverage - since decision coverage does not cover all inside cases, so condition coverage is good to have. Example - decoupling all if statements.

Example code that is being analyzed:

```
int parseint(const char *s){
        int value = 0;
        if(!s) return 0;
        while(!s){
                if(*s >= '0' && *s <= '9')
                        value = value * 10 + *s - '0';
                else
                        return -1;
                s++;
        }
        return value;
}
```

Control Flow Graph (CFG) - represents how to go through the program. Block is either executed or not. Basic block is a sequences of statements and are executed sequentially in the direction of the graph.

It can have branch coverage, path coverage (used by fuzzing)

gcov is a nice tool to get line coverage

Fuzz testing

Testing with random inputs

Correctness is no longer a goal, security is.

Goal: to crash the program.

Usually also tracks coverage to reach as much code as possible.

Generational Fuzzing - specify a grammar for the program to fuzz.
Generate random inputs according to the grammar (for each rule, we pick a random option). This helps to have plausible inputs and generate a lot of sets of tests.

Mutational Fuzzing - no grammar, any byte sequence can be tried.
Genetic algorithm - select most promising inputs (that increase the coverage), and slightly cange promising inputs and retry.
Mutations are simple randomized operations

How to detect a promising input?

1. at first use the seed inputs (normal inputs that become abnormal ones)
2. those improving coverage
3. execution time should be short, such that it would be good in big set as well as quickly crashing and not stalling
4. changing some interesting variable
5. getting closer to an interesting code location

Blackbox fuzzers get no feedback
works on systems the fuzzer has no control over
can execture program many times
needs to try many inputs to improving coverage

Greybox/whitebox fuzzers get feedback from tested program
typically recompile program with special compiler
track information at runtime
Slower executuions due to intrumentation
Select better unput cases

Goal: improve coverage:
Solution - need to flip branches

Use dynamic taint analysis - mark user input as tainted. Then propagate taint and track how it works.

Dynamic taint alaysis

Benefits
targeted mutations, fewer attempts needed to flip branch

Drawbacks
large runtime overhead
some taint is hard to track (implicit data flow) - it will miss particular cases, because it does not realize that there is a taint two steps deeper.

Greybox fuzzers observe program behaviour, but does not attempt to understand it.

Whitebox fuzzers analyze the input.

Symbolic execution:
Goal: find an input to trigger buffer overflow
To do os, we mark inputs as symbolic (they do not take a concrete value, and we remember operations)
Reason about the constraints from the program.

If you want to use advanced analysis techniques (taint tracking or symbolic inputs), normal CPUs don't understand them, and there is a need for virtualization/ hypervisor (eg QEMU)

American Fuzzy Lop - practical fuzzer (basis for AFL++). Has a modified compiler to identify the path.

Investingating Bugs

Assert statements

Address sanitizer adds instrumentation in c/c++ code to detect memory errors. (program crashes if it encounters its bug).

Undefined behaviour sanitizer.

Valgrind - detects memory errors on uninstrumented binaries (comparable to address sanitizer). Has detection for uninintialized reads and memory leaks.

If the crashing input is found what to do?

Run debugger, example, gdb.
In the debugger there are plenty of nice features - run program with breakpoints, watchpoints, stepping, obtain stack trace.

Penetration Testing

either pay a hacker to attempt to break into your system.

there are companies that pay for pen testers to find vulnerabilities - it is deliberate by the developer.

Exploitation

Example - if the program actually run with elevated privileges, how do we get the privileges exactly.

Actually doing the attack requires low-level insights:

- how to find where the return address is stored?
- where to return to?
- how to actually perfore the attack with this information?

Stack layout

to understand the stack layout, it is possible to compile the program to assembly.

Usually the return address is %rbp+8

Question about fuzzing

```
int main(...){
        long i, j = 0, value;
        char input[256];
        if(read(0, input, 256) != 256) return 1;
        memcpy(&value, input + 11, 8);
        for(i = 0; i < value; i++) j+= 1;
        if(j == 42) vuln();
        return 0;
}
```

Q: Does DTA find this fulnerability?
Answer - no, because even though taint is tracked through read, memcpy, but because the branch (vuln()) is reached through for loop, it is not possible to track the taint in the i or j from the value. This is not possible because there are too

many possibilities that the system would need to detect, so thus it is not detected.

Question on assembly

INSERT ASSEMBLY AND C CODE

Q: where is the return value?
Answer - it is on %rbp + 8

Q: where does the user input end up?
Answer - first start with the callq statements and then trace the return values (are stored in %eax/%rax registers) and then trace where it goes afterwards.
There is a scalar that is stored in %rdx, that is shifted at first by 4, which means multiplying by 16 to accommodate for the struct.
The buffer is stored in -0x230(%rbp), so to overwrite the return address we do (%rbp + 0x8) - (%rbp - 0x230) = 0x230. We write 0x23 (35base10) structs (0x230 / 0x16) and add 0x8 to have the offset inside the struct.

Q: what input do we give?
Answer - we give 35 to have 35 structs and then provide shellcode: Args: *35 0 0 shellcode*

Attacker goal

Assume the attacker found the vulnerability to overwrite the return address and how to reach that address.

Where to we point the return address to? - keep in mind that we can only execute code in the same process.

There are two options:

- Injecting code (in modern systems impossible)
- Reusing code

x86 CPUs do not distinguish code and data, so if memory permissions allow we can read program code as data and we can execute data as if program code.

To inject:

- specify as a parameter
- specify as an env variable
- get it from the memory

Injected code must: work regardless of where it is stored in the memory, not depend on any external code such as libraries and not contain NULL bytes, thus it has to be assembly. And do something that gives attacker control of the system.

Goal - start a shell under these constraints.

To do this, we need to use a system call to start a shell.

We do the execve call (load a program in a current process). (execve("/bin/sh", argv, NULL) and char[] argv = {"/bin/sh", NULL}. This is necessary to have 2 times "/bin/sh" to get correct result).

How to do this without shared libraries (libc)?
Get the assembly code from the objdump. It can be seen that syscall does not take any new parameters to find the name of the program, but the parameter registers are the same as execve call.

INSERT ASSEMBLY CODE FROM THE SLIDES OF EXECVE

Shellcode needs the following:

- have a string "/bin/sh" in memory
- an array in memory, containing a pointer to "/bin/sh" and a NULL ptr
- a ptr to the string in %rdi (program name)
- a ptr to argv
- a syscall

How the shellcode works:
leaq to get a location of the string in the shellcode relative to the %rip, because we cannot have a static address. Then manually put the null terminator to the end of the string. Set argv[0] to "AAAAA" and argv[1] to "BBBBB". Set argv. set envp. set syscall number and go to kernel (no return).

There still are problems with the shellcode:

- it still contains null bytes $0when we do movq 0$ to create argv parts
- %rax is 64 bytes and %eax has null bytes at the start when converted for the syscall
- leaq has only 32 bit arguments, thus %rip will be converted with some null bytes.

To fix this - we move string to be before the shellcode (address is negative). Then have xorl %eax with itself to add a new byte in %rax thus making zeroes in the first part. We use %rax (zeroed out part) to put null bytes in the args. Then we use only %al to have the argument for syscall.

Injecting shellcode
since environment is predictable, it is nice to inject code through it.
in dumping the program it is possible to notice that there is null bytes padding before the stack begins.
So to get the injection - skip 8 bytes (-8), skip program name length and skip the shellcode length.
To get the correct address we need to get it is needed to do static cast to the pointer as long ptr. (with an offset 24 to get to the executable part).

another example
Problem there lies in the fact that strlen does not account for the null byte, so there can be 48 chars but the null byte would be 49th, thus not being copied by strcpy.

Pointer is just an integer storing a memory address.
the x86_64 arch stores pointers as 64bit integers. However it uses only 48 least significant bits are actually used. The 16 msb are null bytes (warning).
However x86_64 arch is little endian - lsb is stored first (looks reversed). This means the null bytes of the pointer are stored at the end, which is good.

2 Lectures skipped - web security

Questions (Web security)

Q1: You add security checks to ensure user-submitted content can only be retrieced by the same user that submitted it. This completely mitigates stored XSS attacks - True. If we submit the content ourselves, it will be stored for us and only sent back to use. Hence, we cannot unject js in other users' sessions, and there is no point in doing so in our own.

Fine-grained CFI is the strictest we can do statically. It is better for the forward-edge.

Shadow stack is not static, since it traces the dynamic part. However it cannot do indirect calls.

For user input there should be full validation. Malloc should also be handled if it does not return correctly, we may not have a working exit, but it should show an error message.

Q2: when a user creates a session, you store the remote IP address in your database. On any subsequent requests, you create a new session if the IP address does not match the value stored for the session.
Q: what attack does this mitigate? - Session fixation attack
Q: which drawbacks does it have? - breaks on mobile devices that switch between conns; cannot distinguish systems on the same external IP address (NAT)
Q: can it be bypassed? - only within a network with a common ext IP address, or if we can spoof the IP address

SQL Injections

Adding "' OR 1=1;--" is a common way to do an SQL injection. Or doing a UNION to retrieve the data from other tables.

It is often blind - client does not know the output for every input.

Mitigation - do not trust any input SQL code. There can be a second order SQL queries, so to mitigate, use prepared statements and procedures.

Cross-site request forgery (CSRF)

Generally the stuff that is not shown, it is assumed that is secure

Stored CSRF attack is a way to attack user data through browser unintentionally makes a request of logged in user.

It is hard to prevent even with proper escaping

Reqs: attacker can get target to visit a URL. Visiting URL is sufficient to change persistent state (leaks are not posssible)

There is a possibility to have a non-stored CSRF - cookie is sent based on requested URL and very hard to prevent

CSRF is an example of confused deputy type of attack.

Mitigation - requuire HTTP POST requests for each operation that can change state. This is so that it is harder to change information with CSRF.
Add a random token to each state changing operation and check it - forged reqs will not have correct token. Attackers will not be able to forge this token.

Unrestricted file uploads

in improperly configured server you can upload a file with a wrong extension (e.g. .php), and then exploited by running it.

```
<html><body><php>
$php(shell_exec(GET['cmd']));
</php></body></html>
```

File extension determines how the file is interpreted.

Mitigation: sanitize file names - only allow expected file extensions (allowlist), also we have to be careful with path traversal.

Defence in depth:
web server config: allow only specific dirs for script execution, and disallow writes to script dirs.
Apply principle of least privilege using operating system: configure system to use only executable,

Deserialization of untrusted data

Serialization: take object and convert it into data. The idea is to store the object on the disk or use a stored procedure.

Many languages offer serialization out-of-the-box

Representation of Serialized Objects: JSON, SOAP (based on XML), YAML (superset of JSON); Language-specific: Python-Pickle, Java Object Serialization

Deserialization danger: call the subprocess constructor and provide the program (e.g. Python "subprocess.POpen"), thus executing the program.

Encryption could help in a sense that it ensures that it comes from a trusted source.

This allows arbitrary code execution.

SSRF

server downloads content that is in the url, therefore SSRF vulnerability is there iff there is urls submitted.

Exam

Q1
a) addition is to compute the entropy, multiplication is to compute the

combinations
b) No, because it depends on the backward-edge or forward-edge necessity in particular context.
c)
d) Second pre-image resistance - . (NOT CONFUSE WITH COLLISION RESISTANCE). And then specify why it matters to the signatures
e) What is known plaintext attack. What is known cyphertext attack.
f) What is XSS. If `<script>` tags are prohibitted, what are the possibilities for XSS.

Q2
there is a simple program, find all the issues (vulnerabilities and not)
All the parts that are not shown, are assumed to be safe.
Points are calculated: 3pts for correct problem, -1pt for incorrect, 0pt for duplicate

Q3
a) stack layout - function properties, variables.
b) if shellcode is runnable, specify the payload.
c) one of the defences that were discussed: explain how this mitigates the exploit and if it is possible to change it to still work.

Q4
answer contains the parameters that cover some code that others can't.

Q5
encryption - detect and decrypt, and have the final answer.

Q6
with small information talk about the web security vulnerabilities.